University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Pitt SCI

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - **Lab 10**: ~~Tuesday 4/11~~ May 1 @ 11:59 pm
  - **Lab 11**: ~~Tuesday 4/18~~ May 1 @ 11:59 pm
  - **Lab 12**: May 1 @ 11:59 pm
  - **Homework 11**: ~~Friday 4/14~~ May 1 @ 11:59 pm
  - **Homework 12**: May 1 @ 11:59 pm
  - **Assignment 4**: ~~Friday 4/14~~ May 1 @ 11:59 pm
    - Support video and slides on Canvas + Solutions for Labs 8 and 9
  - **Assignment 5**: May 1 @ 11:59 pm
    - to be posted tonight

Final Exam

- **Friday 4/28 12:00-13:50**
  - 169 Crawford Hall
- Same format as midterm
- **Non-cumulative**
- Study guide and practice test on Canvas
- **Review Session** during Finals' Week
  - Date and time TBD
  - recorded

Bonus Opportunities

- **Bonus Lab**
  - worth up to 1%
  - lowest two labs still dropped
- **Bonus Homework**
  - worth up to 1%
  - lowest two homework assignments still dropped
- bonus point for class when

  OMETs response rate >= 80%
  - Currently at 12%
  - Deadline is Sunday 4/23

Previous Lecture

Dynamic Programming examples:

- ○ Longest Common Subsequence
- ○ Reinforcement Learning

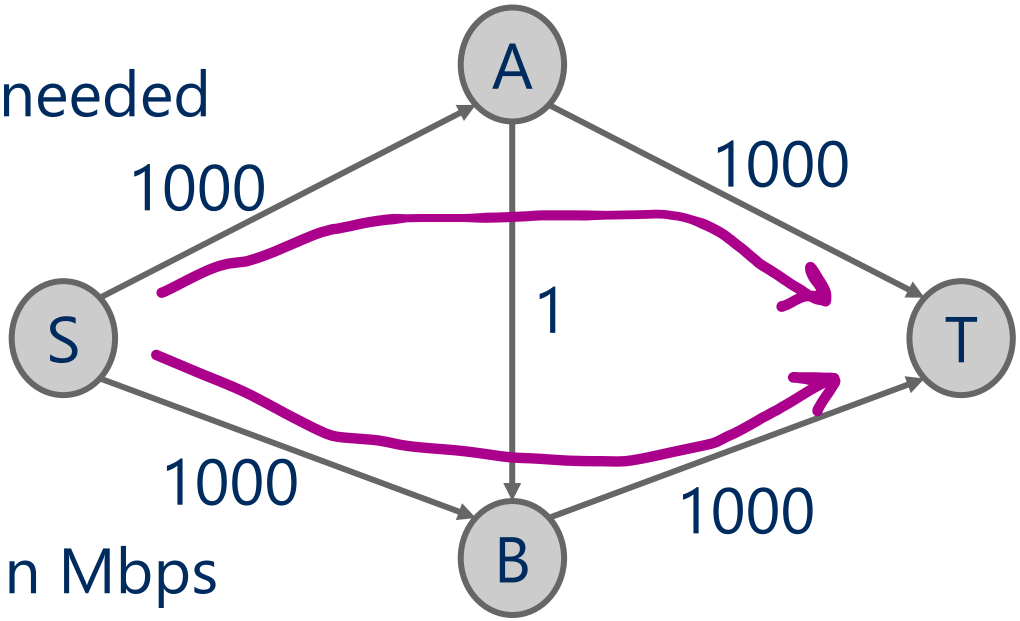
- **Maximum Flow** Problem
  - ○ Ford Fulkerson Framework

This Lecture

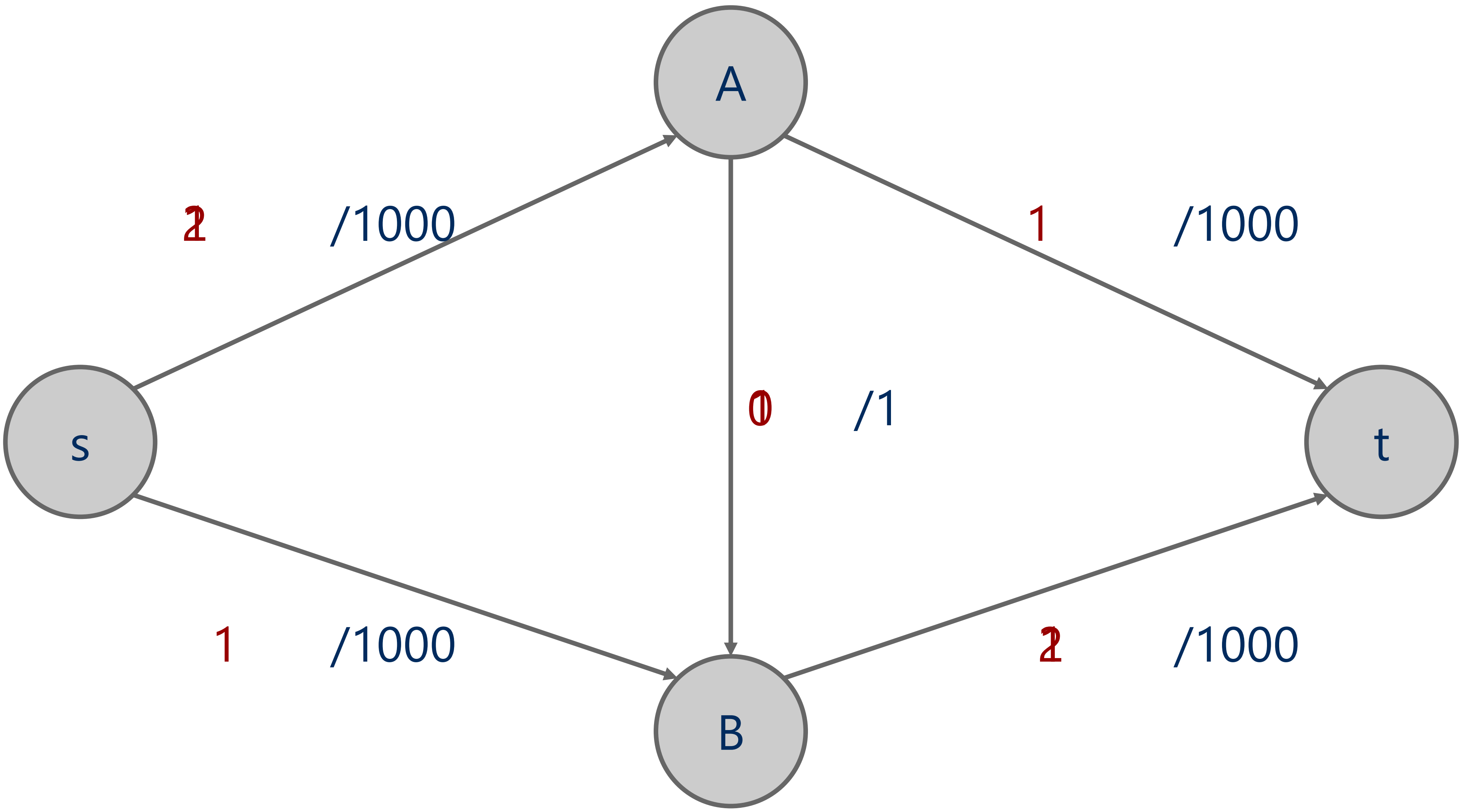
- **Maximum Flow** Problem: useful for general problem solving
  - Edmonds Karp
  - Push Relabel
  - An application of Maximum Flow
- Graph compression
- Local Search

Problem of the Day: Finding Bottlenecks

- send a large file from S to T over a computer network
  - as fast as possible
  - over multiple network paths if needed
- Input:
  - computer network
    - nodes and links
    - links labeled by link speed in Mbps
  - nodes A and B
- Output:
  - The **maximum network speed** possible



Worst-case Runtime for Ford-Fulkerson



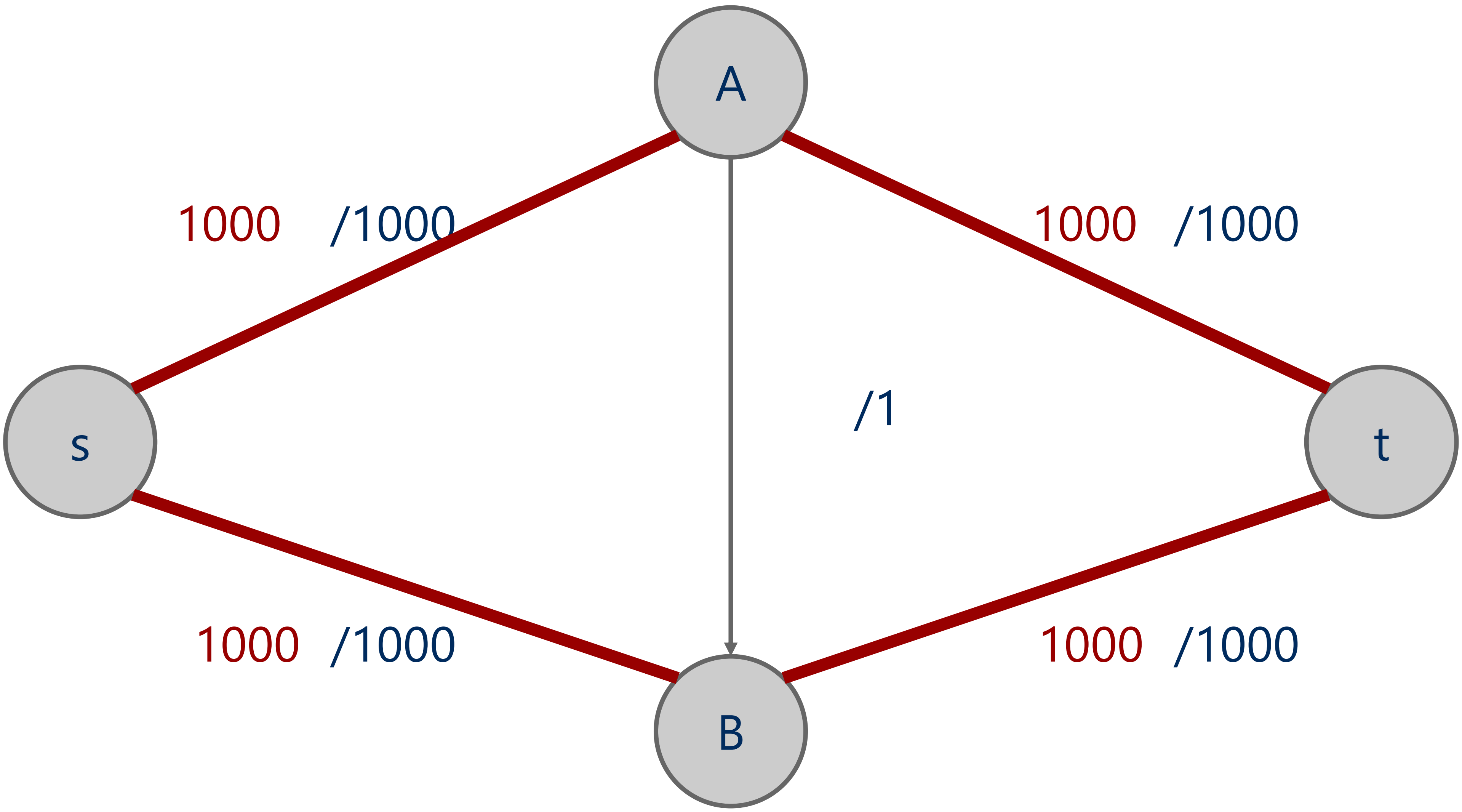
Worst-case Runtime of Ford-Fulkerson

- O(f * (e+v))
  - f: value of maximum flow
  - e+v: time to find an augmenting path
- is that **polynomial** in the ***input size***?
- **No!** f is exponential in **bit length** of f
- Runtime is $O(2^{|f|} * (e+v))$

Edmonds Karp

- How the augmenting path is chosen affects the **performance** of the search for max flow
- Edmonds and Karp proposed a **shortest path heuristic** for Ford Fulkerson
    - Use **BFS** to find augmenting paths

Edmonds Karp using BFS to find augmenting paths



Edmonds Karp

- Running time is $O(e^2 v)$
    - **Proof:** Proposition G in Chapter 6

But our flow graph is weighted...

Edmonds-Karp only uses BFS

- BFS finds spanning trees and shortest paths for **unweighted** graphs
- some **weight-based** measure of priority to find augmenting paths?

Maximum Capacity Path

- Proposed by Edmonds and Karp
- implemented by modifying Dijkstra's shortest paths algorithm
- Define **flow[v]** as the maximum amount of flow from s → v along a **single path**
- Each iteration, set curr as an unmarked vertex with the largest flow
- For each neighbor w of curr:
  - if **min(flow[curr], residual capacity of edge (curr, w))** > flow[w]
    - update flow[w] and parent[w] to be curr

Flow edge implementation (Bonus Lab)

- For each **edge**, we need to store:
  - **from** vertex
  - **to** vertex
  - **edge capacity**
  - **edge flow**
  - residual capacities
    - For **forwards** and **backwards** edges

FlowEdge.java

```
public class FlowEdge {
   private final int v;              // from
   private final int w;              // to
   private final double capacity;    // capacity
   private double flow;              // flow
     …
     public double residualCapacityTo(int vertex) {
      if      (vertex == v) return flow;
      else if (vertex == w) return capacity - flow;
      else throw new
       IllegalArgumentException("Illegal endpoint");
   }
   …
}
```

BFS search for an augmenting path (pseudocode)

```
edgeTo = [v]
marked = [v]
Queue q
q.enqueue(s)
marked[s] = true
while !q.isEmpty():
    v = q.dequeue()
    for each edge in AdjList[v]:
        if residualCapacity(other end-point) > 0:
            if !marked[w]:
                edgeTo[w] = v;
                marked[w] = true;
                q.enqueue(w);
```

Each FlowEdge object is stored in the adjacency list twice:

Once for its forward edge
Once for its backward edge

Push-Relabel Algorithm for Max Flow

- More **efficient** than Edmonds-Karp that uses BFS
  - Running time: **Theta(v^3)** vs. Theta(e^2v)
- **Local** per vertex operations instead of global updates
- Each vertex has a **height** and **excess flow** value

Push-Relabel Algorithm for Max Flow

- **push operation:**
  - Flow **pushed** from higher vertex to lower neighbor
    - height difference of 1 or more
    - over an edge with **residual capacity > 0**
- **relabel operation:**
  - If a vertex's excess flow > 0 and has no lower neighbor
    - **relabel** vertex's height to
      - **1 + min height of neighbors able to receive flow**

Push-Relabel Algorithm for Max Flow

- **push operation:**

- **relabel operation:**

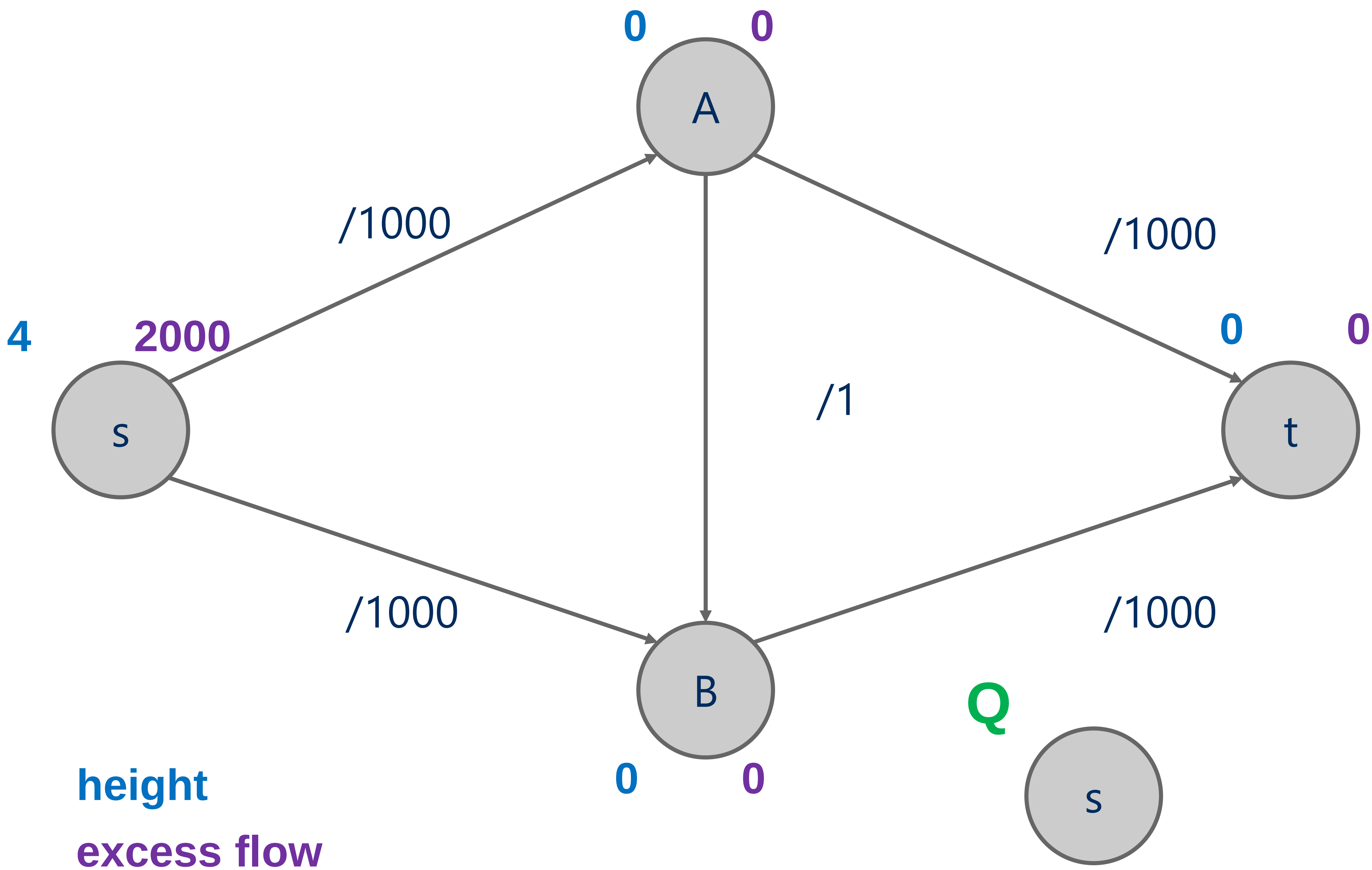
- **Repeat** relabel and push operations until

    - all vertices except source and sink have **0 excess flow**

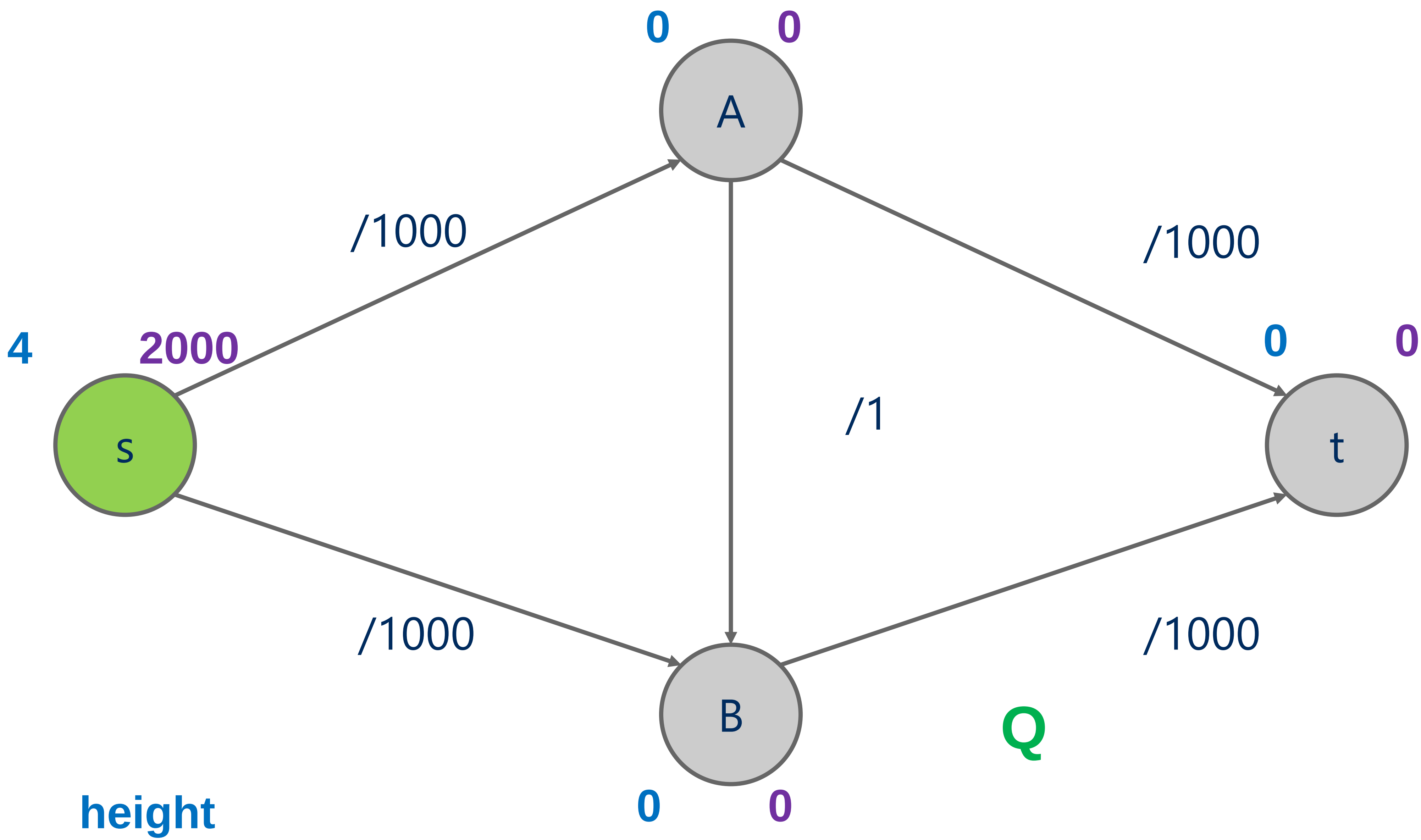
Push-Relabel Algorithm

- height = 0 and excess = 0 for all vertices
- excess[s] = sum of edge capacities out of s
- height[s] = v
- insert s into Q
- **while Q not empty**
    - v = pop head of queue
    - relabel v if needed
    - **for each neighbor w of v:**
        - push as much of v's excess flow to w
        - increase w's excess flow by the pushed amount
        - add w to Q if not already there

Push Relabel Example



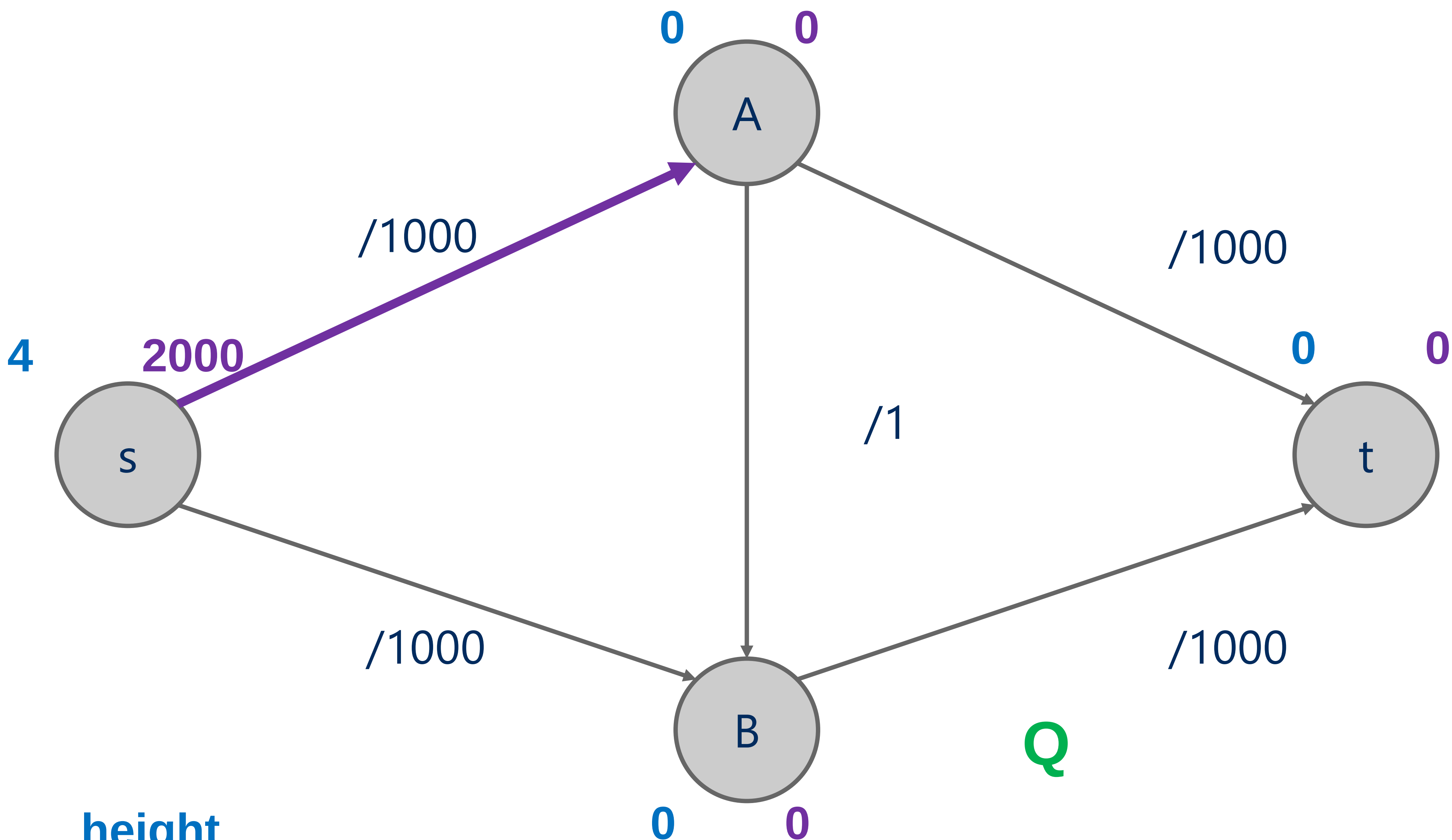
Push Relabel Example



height

excess flow

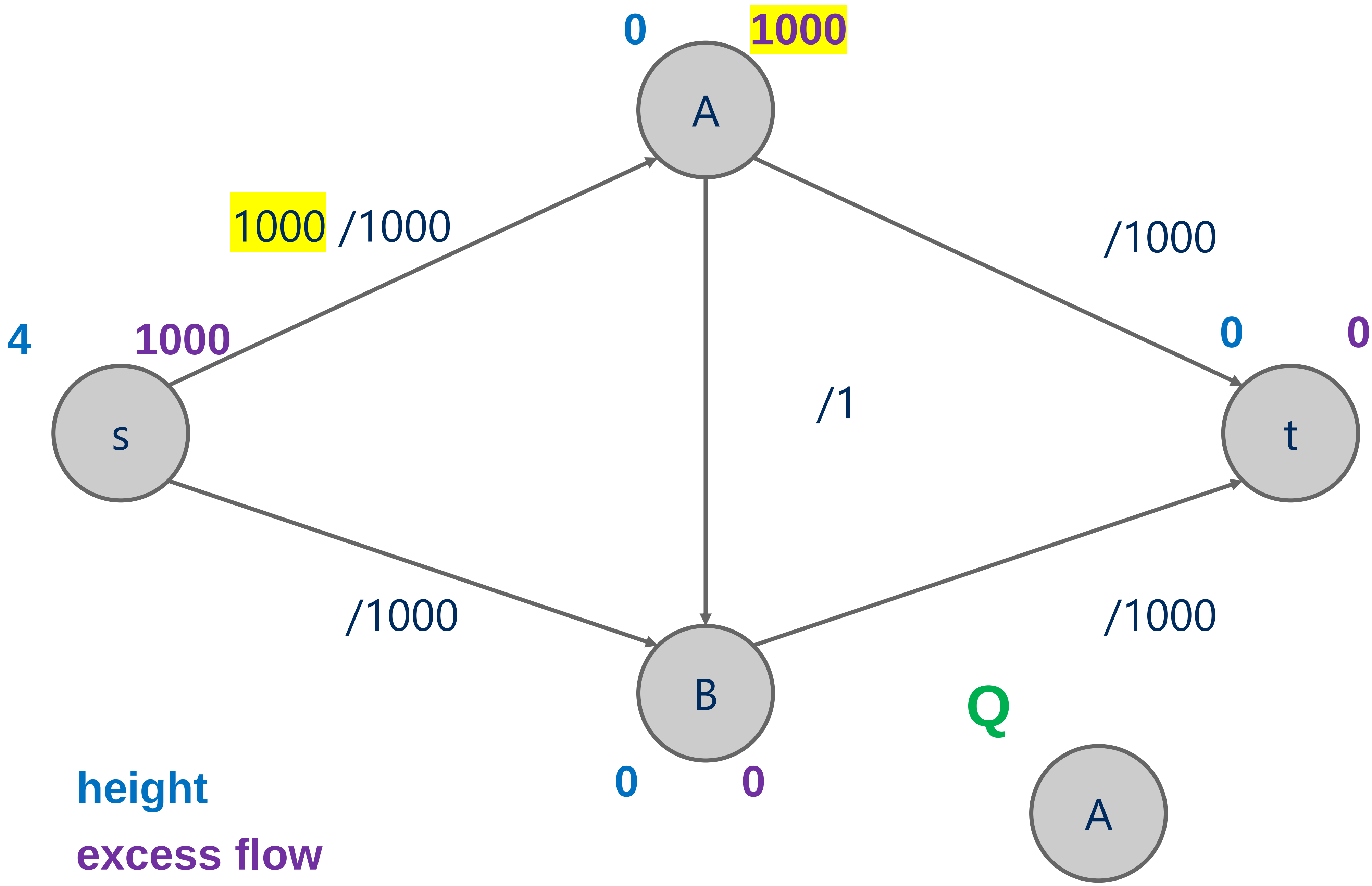
Push Relabel Example



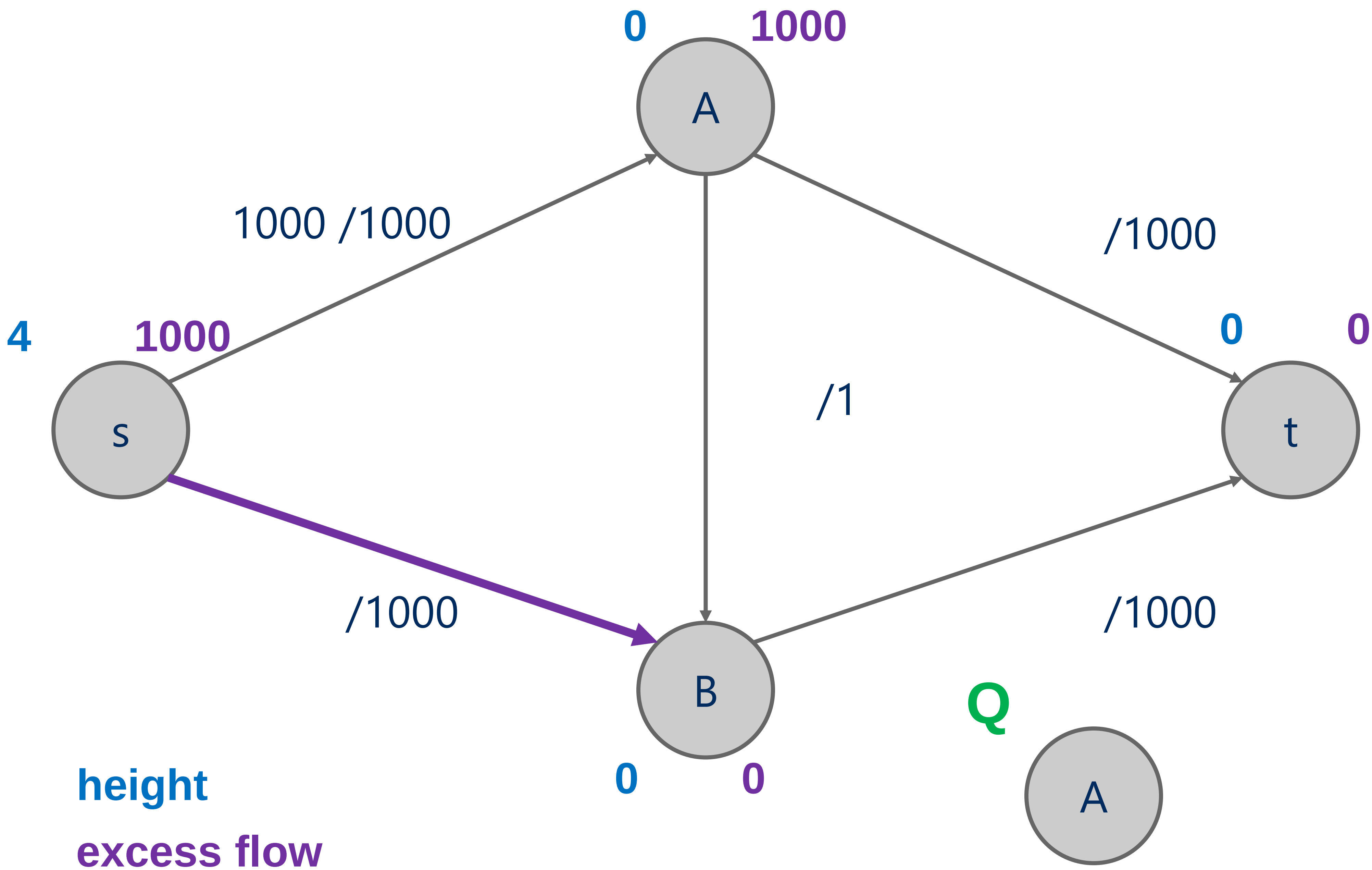
height

excess flow

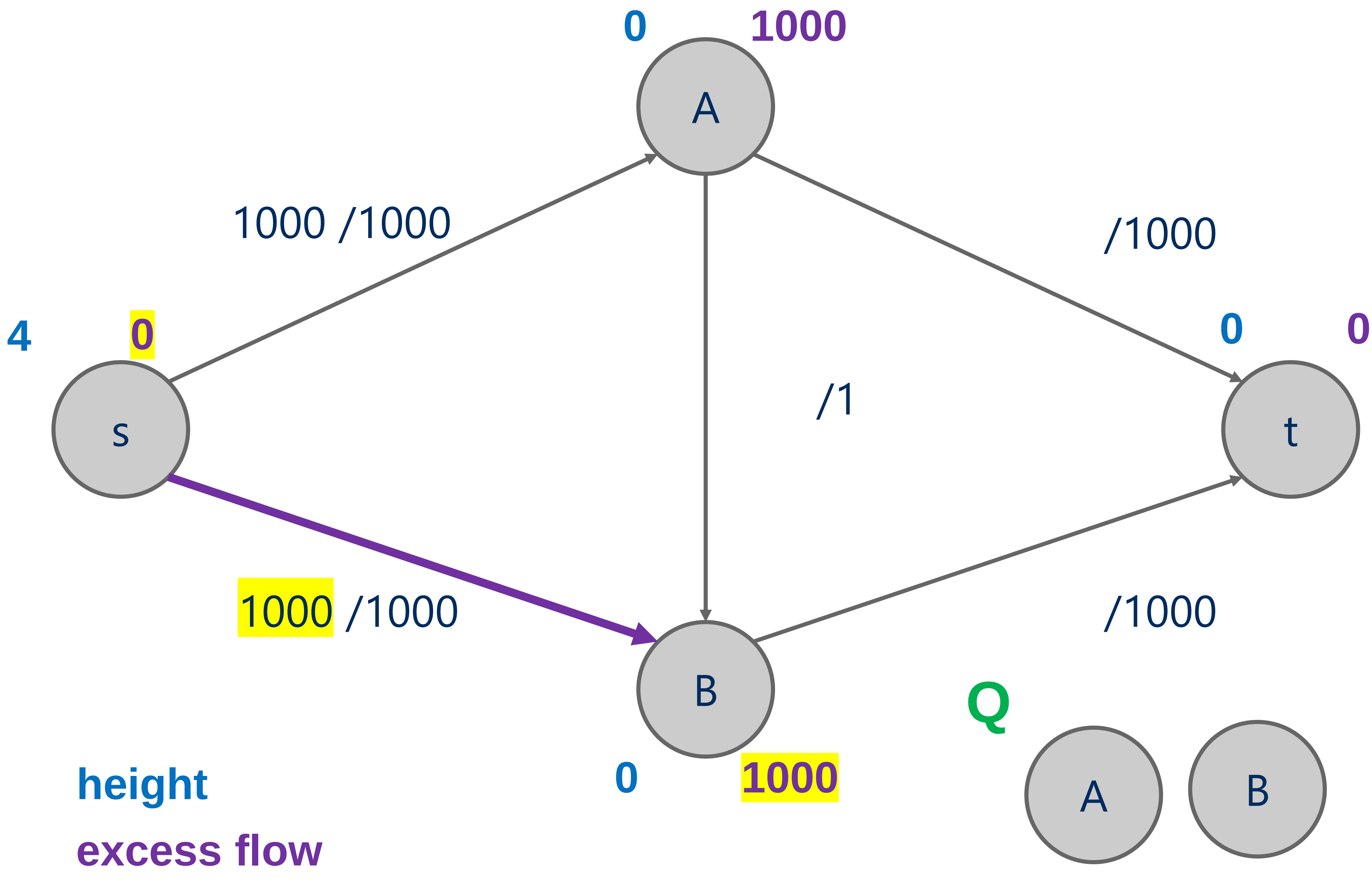
Push Relabel Example



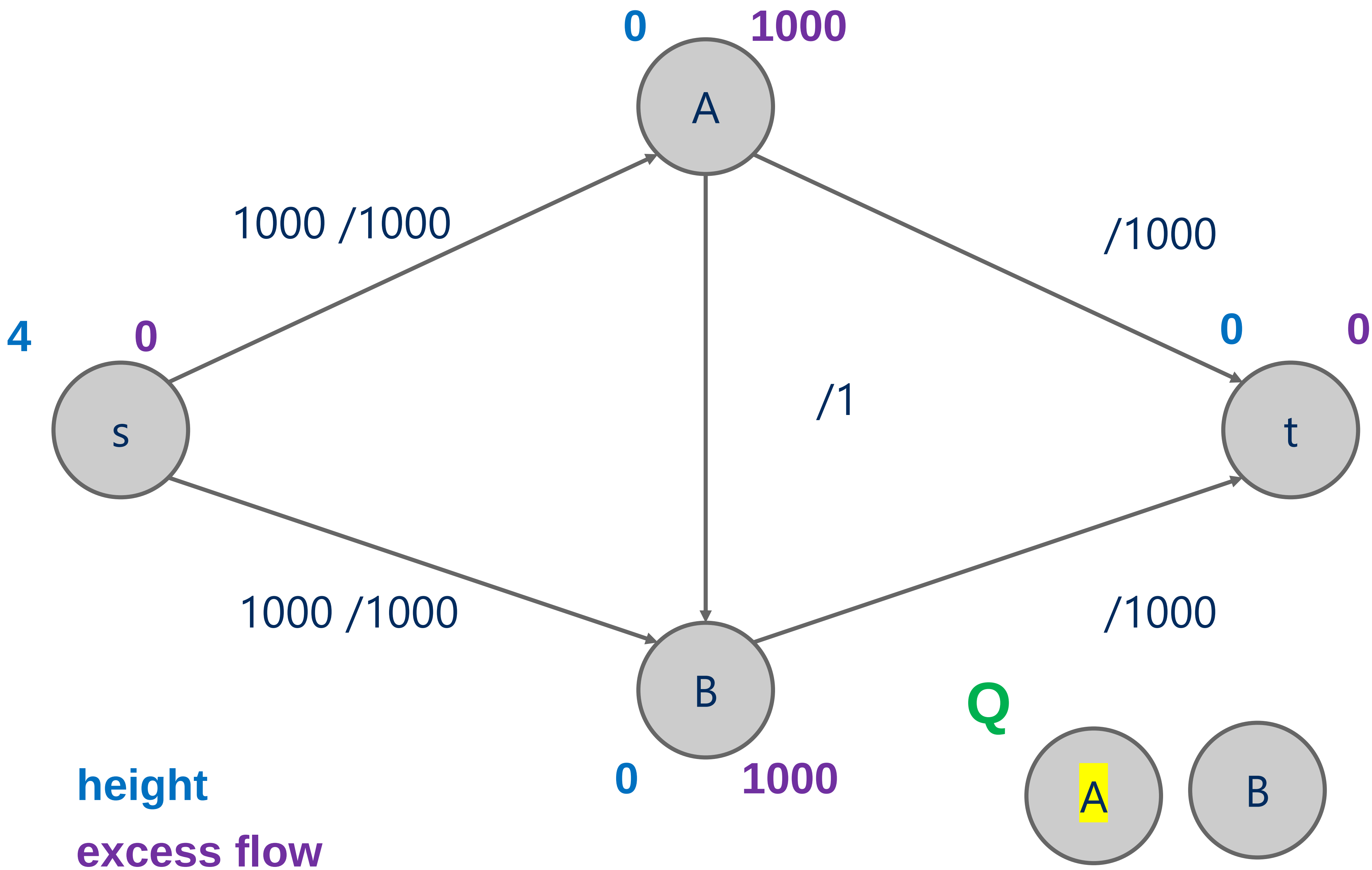
Push Relabel Example



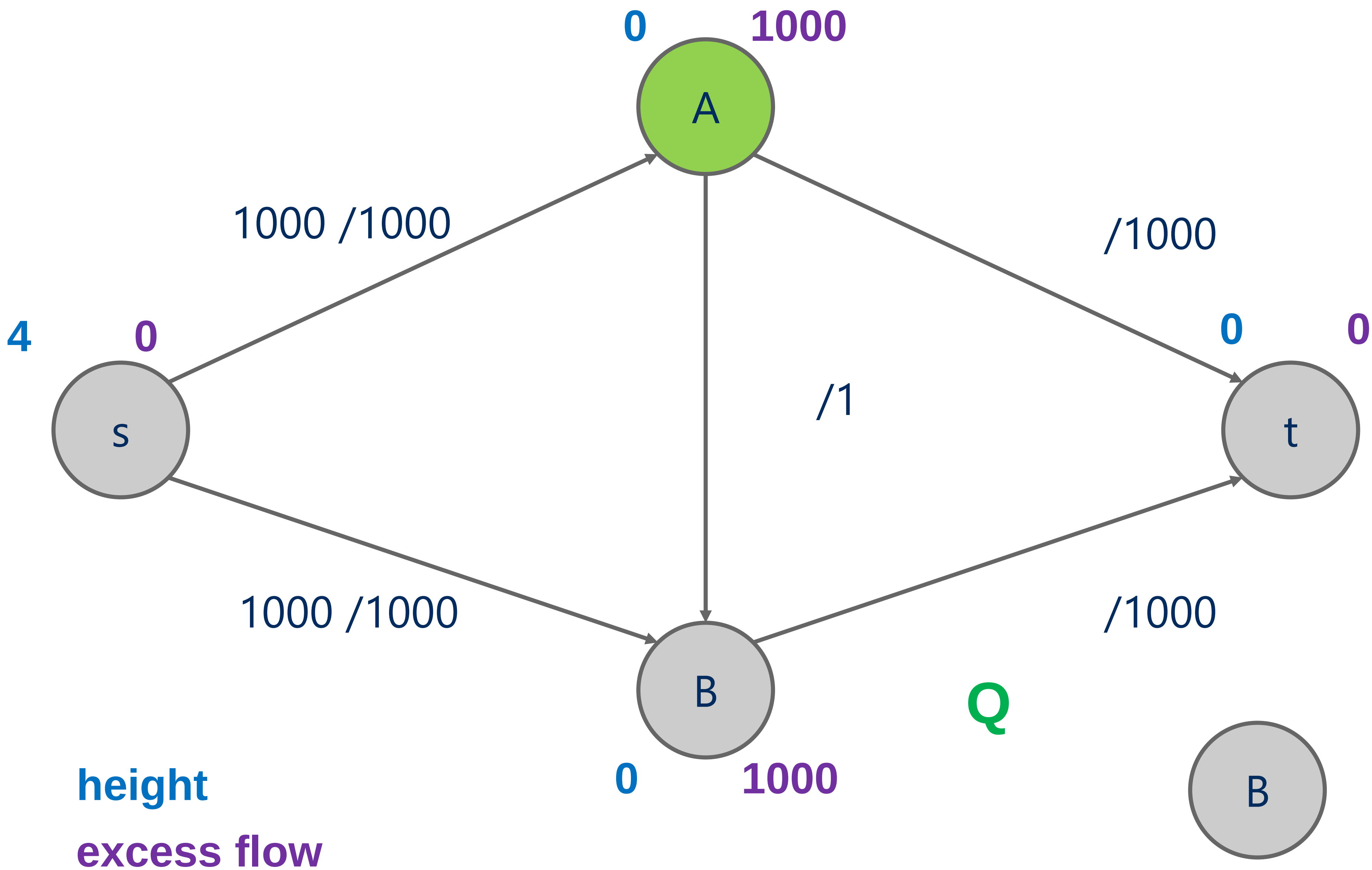
Push Relabel Example



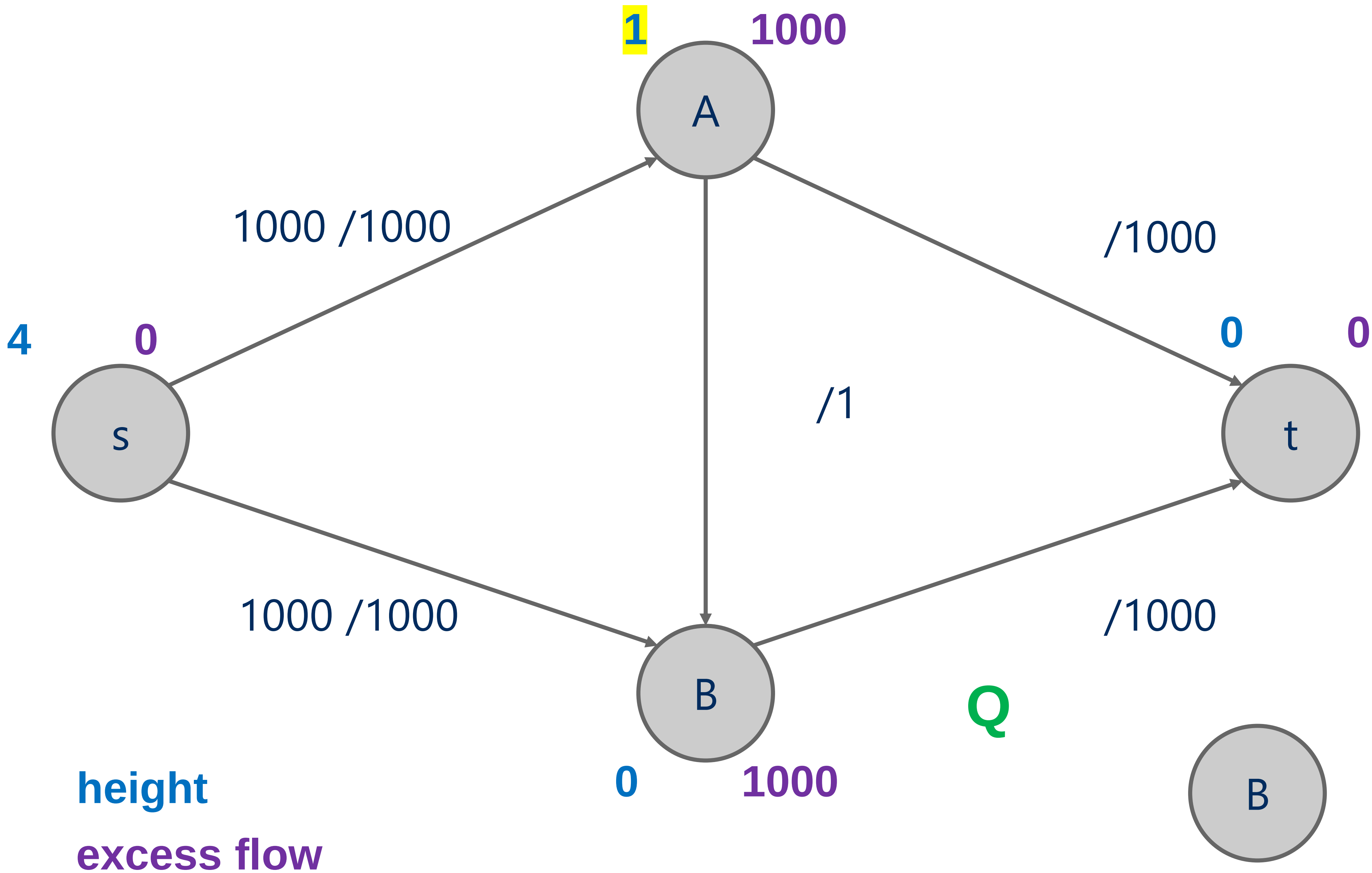
Push Relabel Example



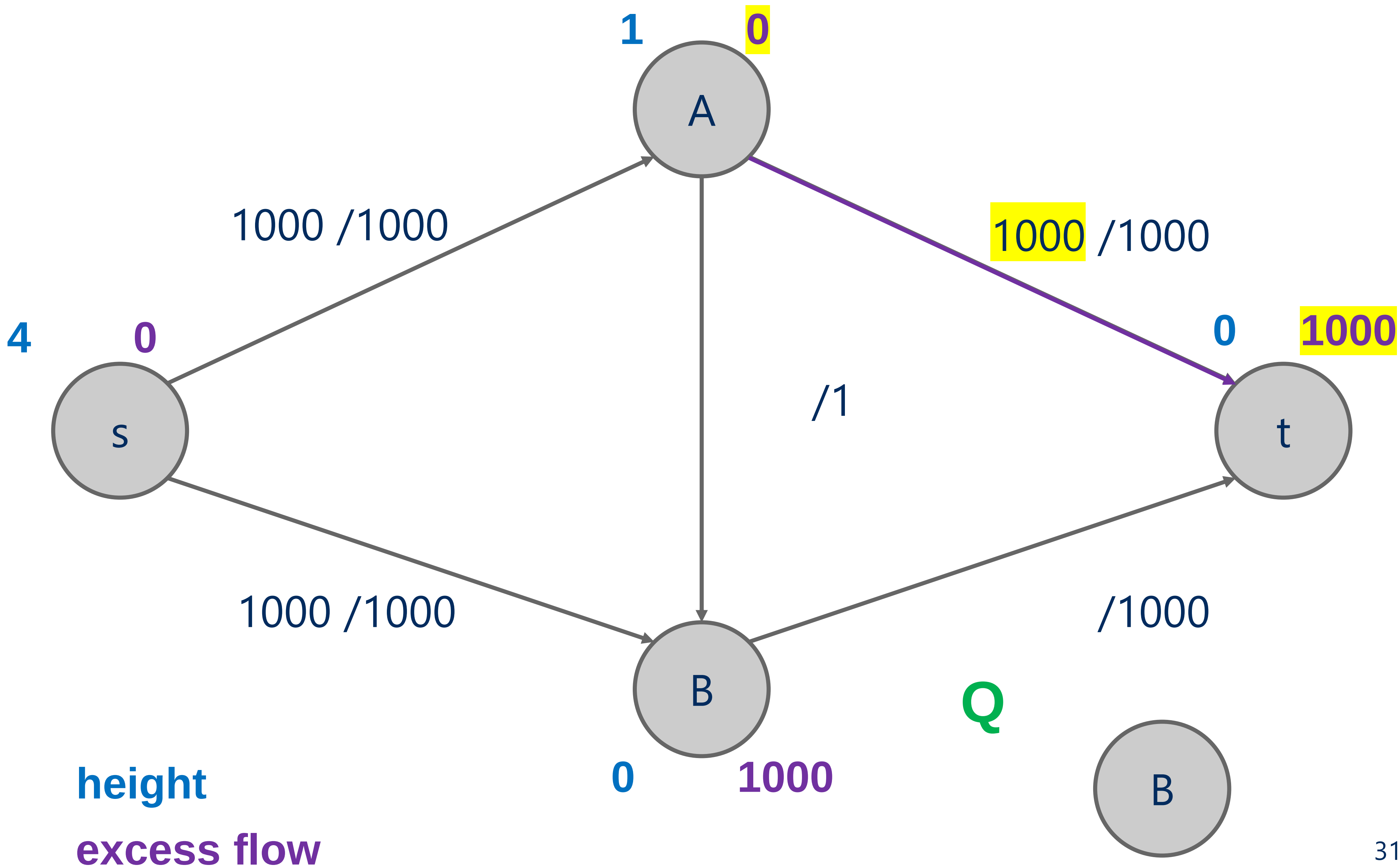
Push Relabel Example



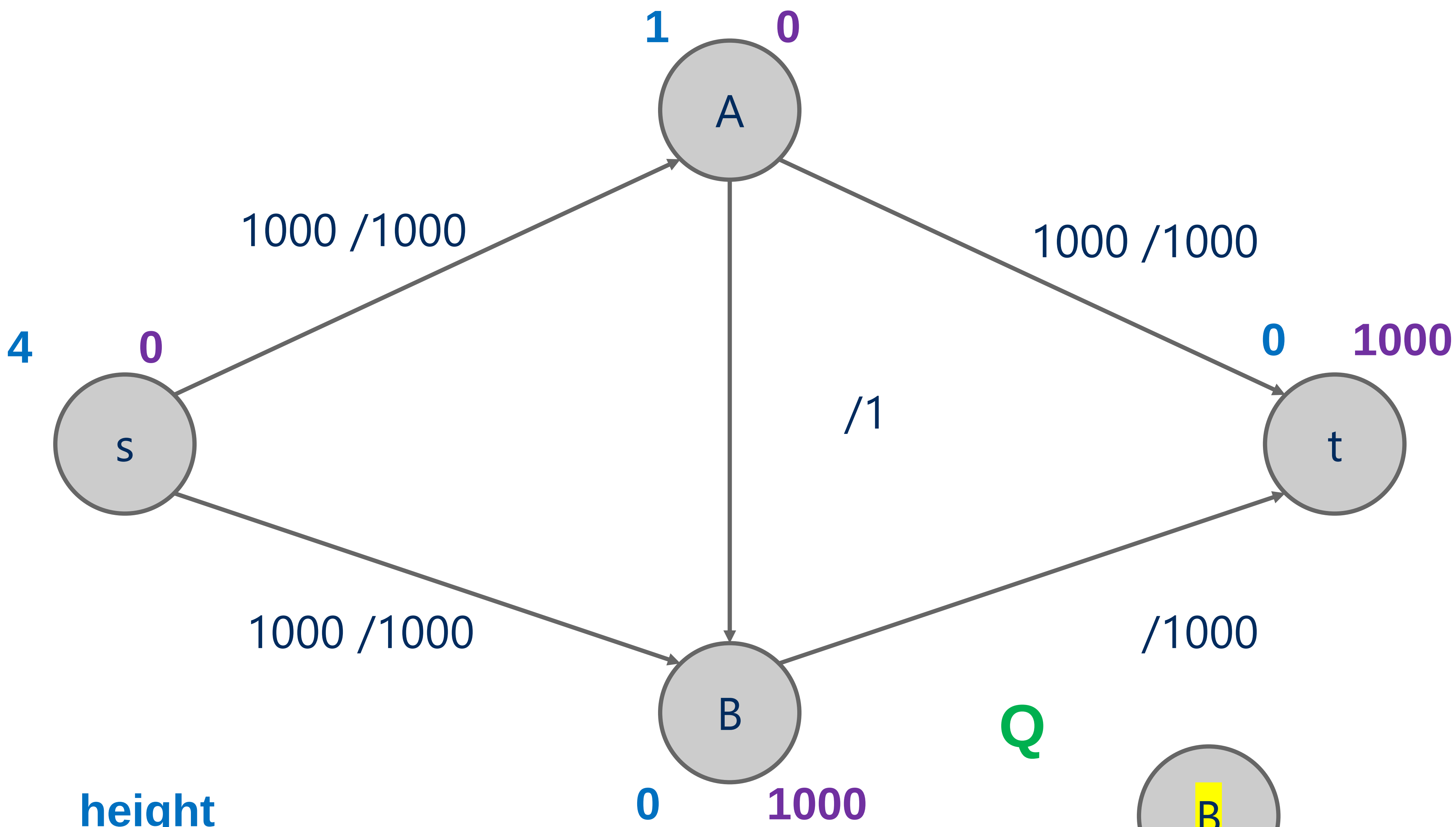
Push Relabel Example



Push Relabel Example



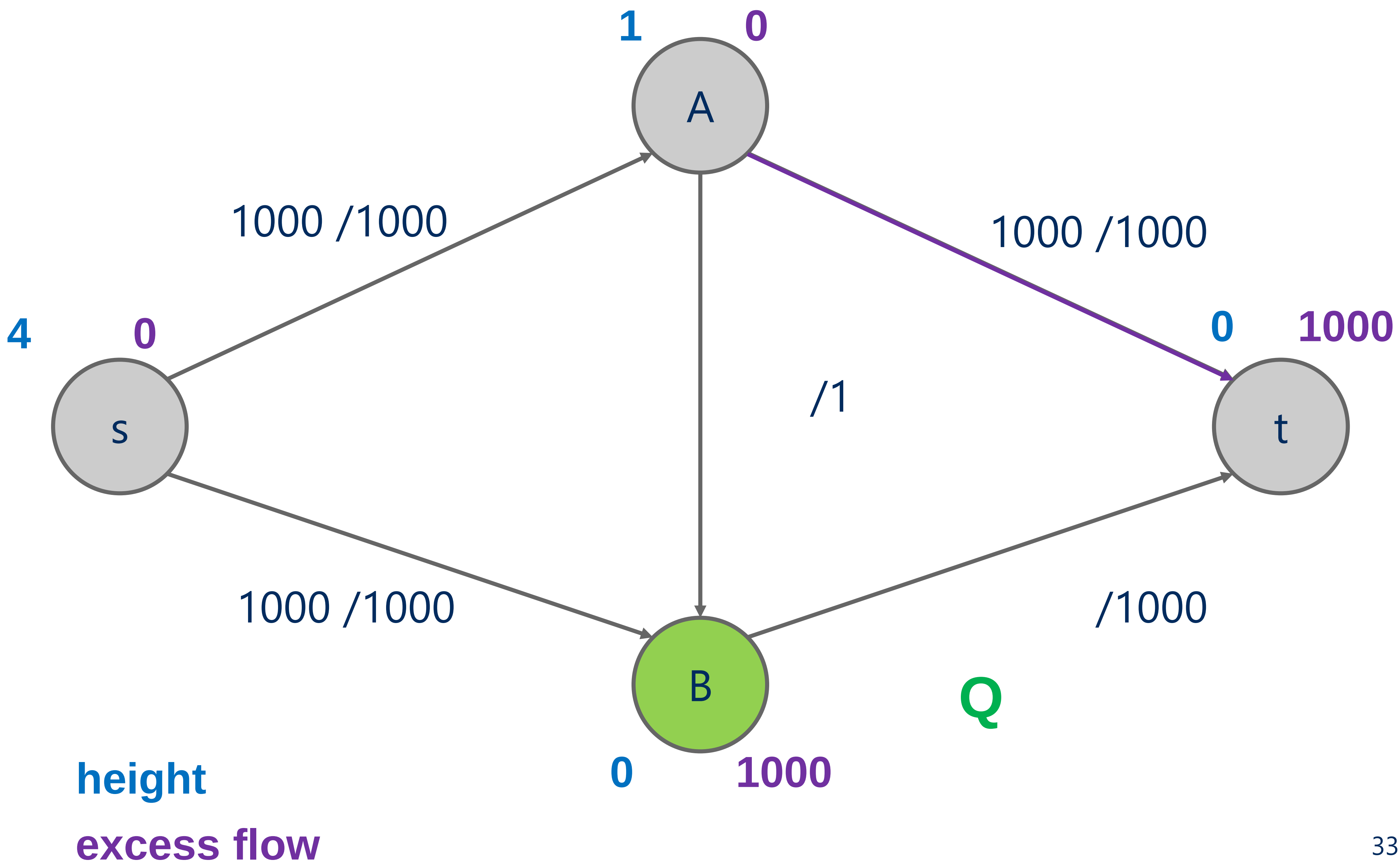
Push Relabel Example



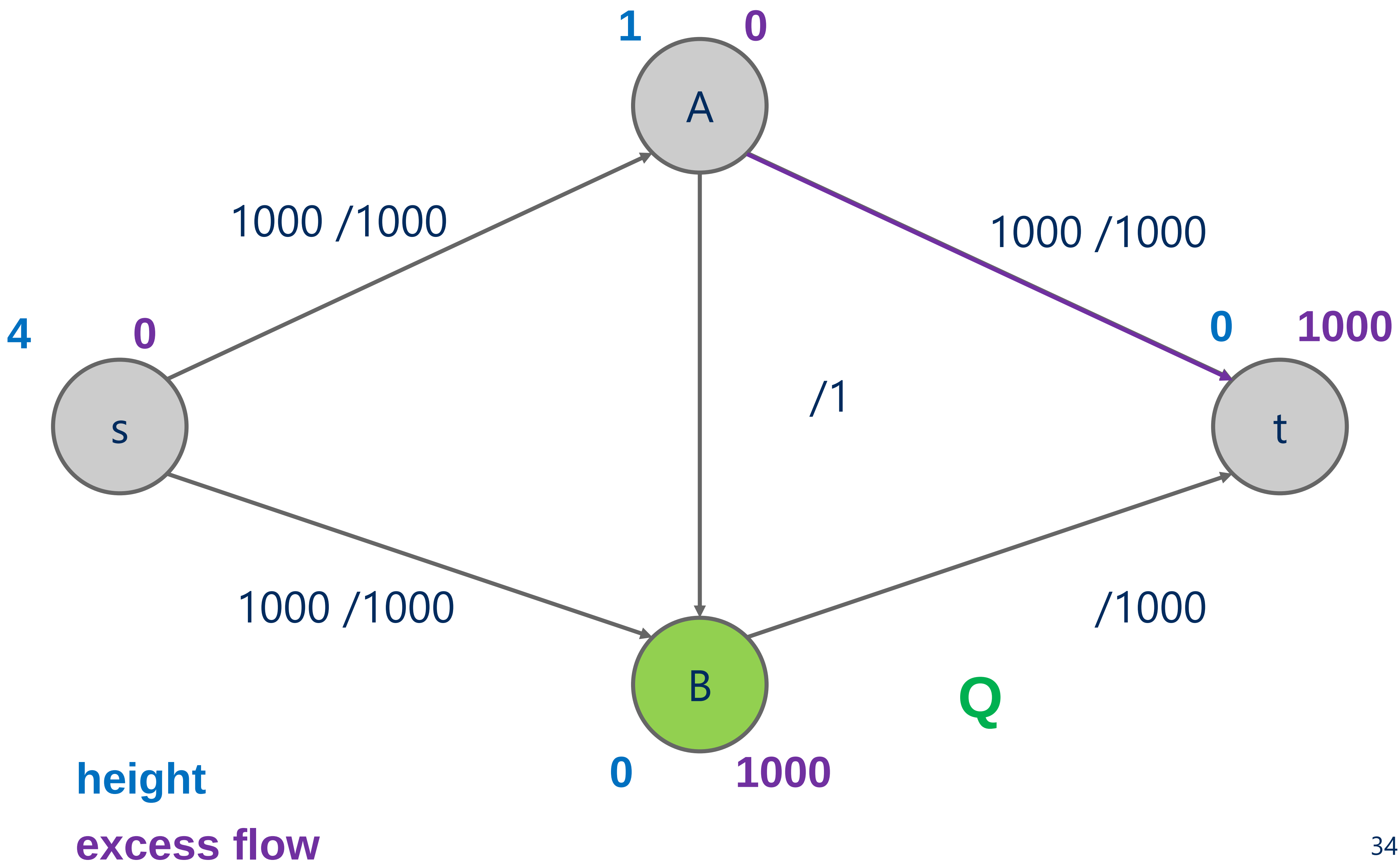
height

excess flow

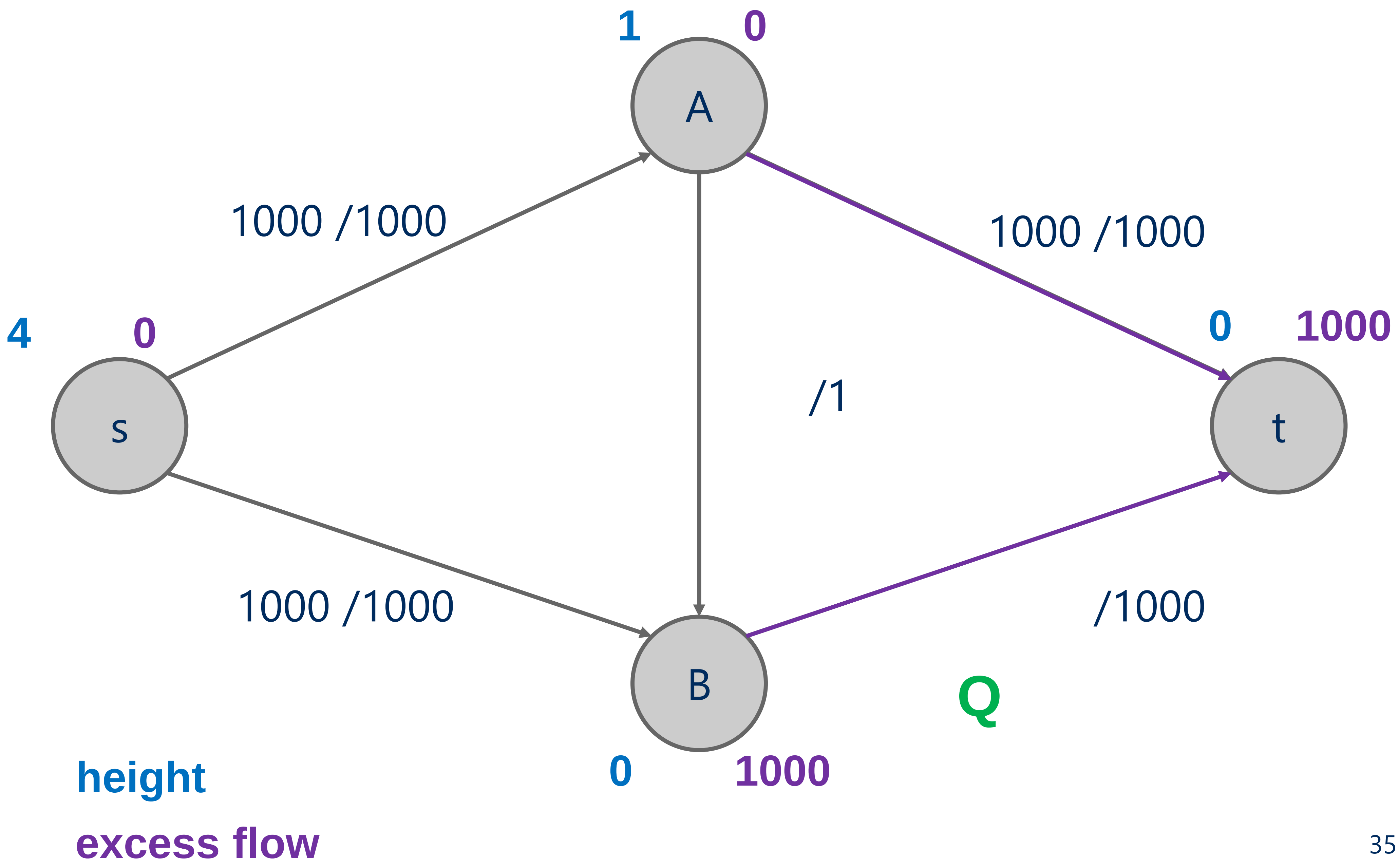
Push Relabel Example



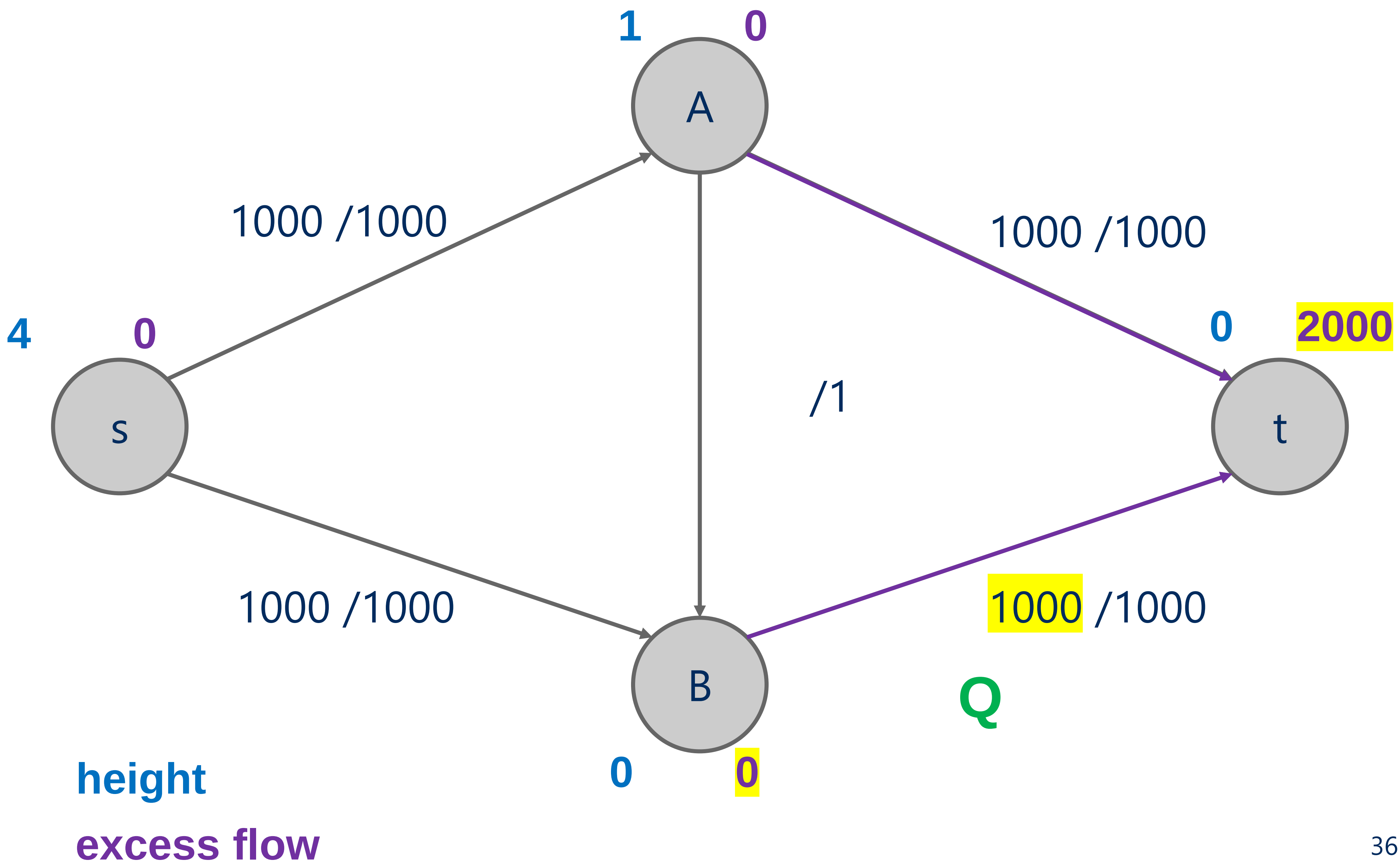
Push Relabel Example



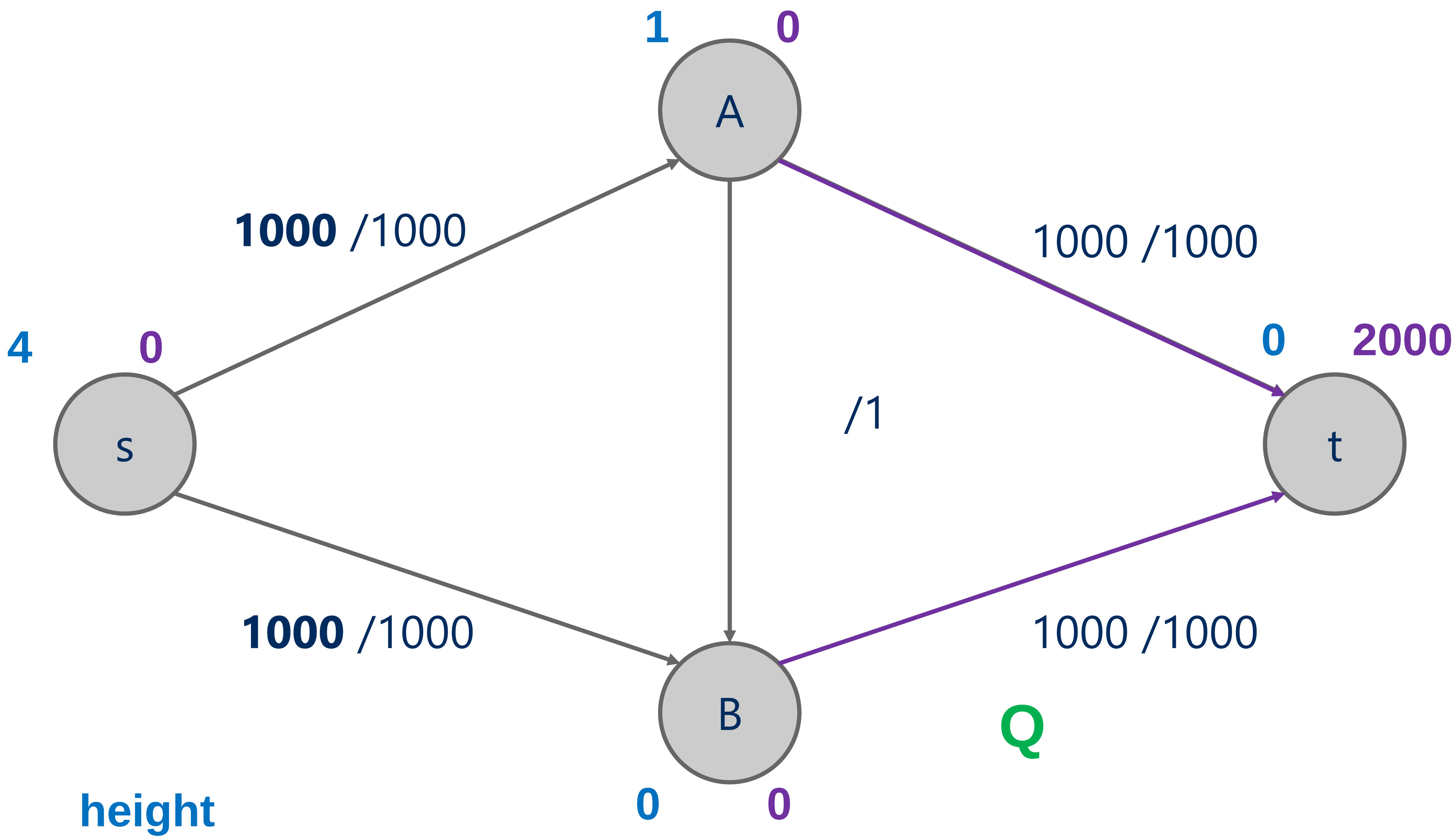
Push Relabel Example



Push Relabel Example



Push Relabel Example

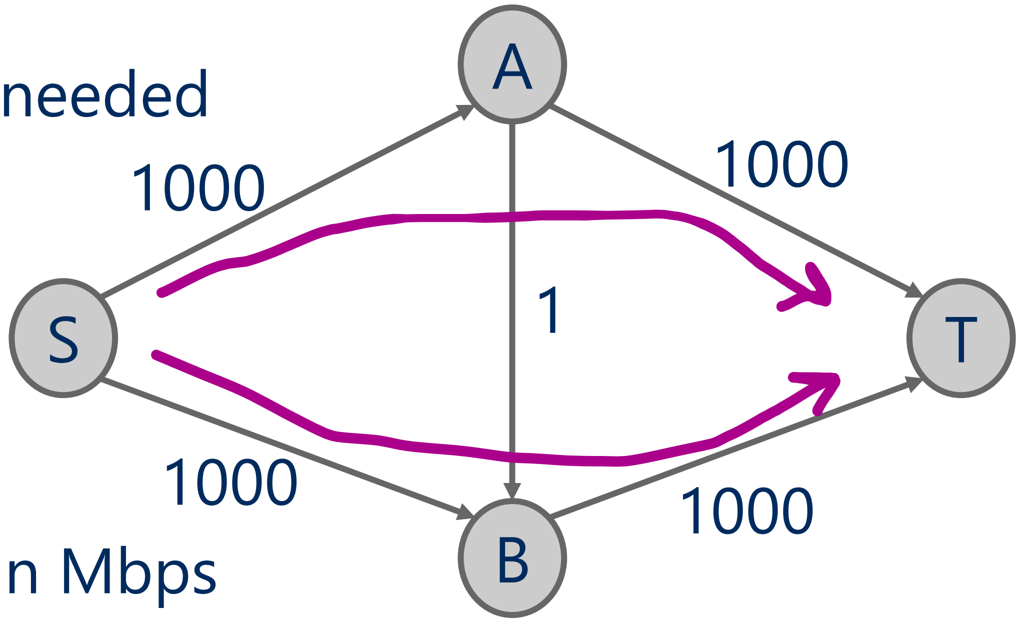


height

excess flow

Problem of the Day: Finding Bottlenecks

- send a large file from S to T over a computer network
  - as fast as possible
  - over multiple network paths if needed
- Input:
  - computer network
    - nodes and links
    - links labeled by link speed in Mbps
  - nodes A and B
- Output:
  - The **maximum network speed** possible

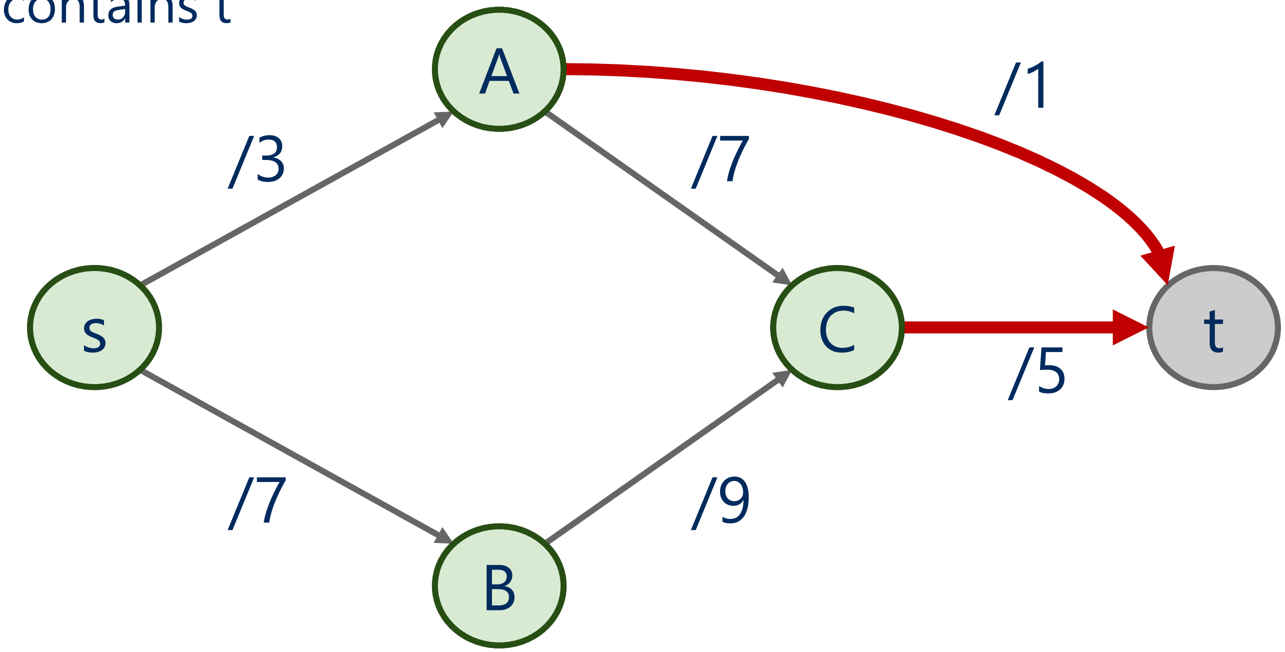


Follow-up Problem

- So, now we found the bottleneck **value**, but which **edges** define the found bottleneck?
    - *Why would you want to know **bottleneck edges**?*

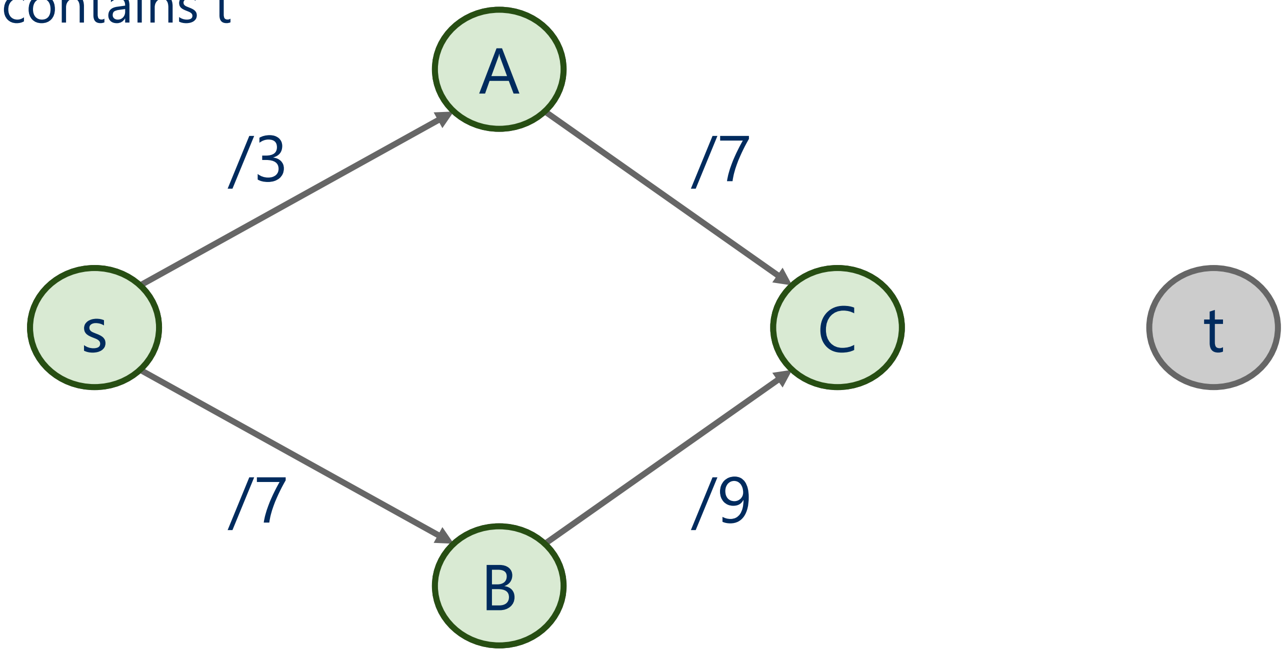
Finding bottleneck edges

- An **st-cut** on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t



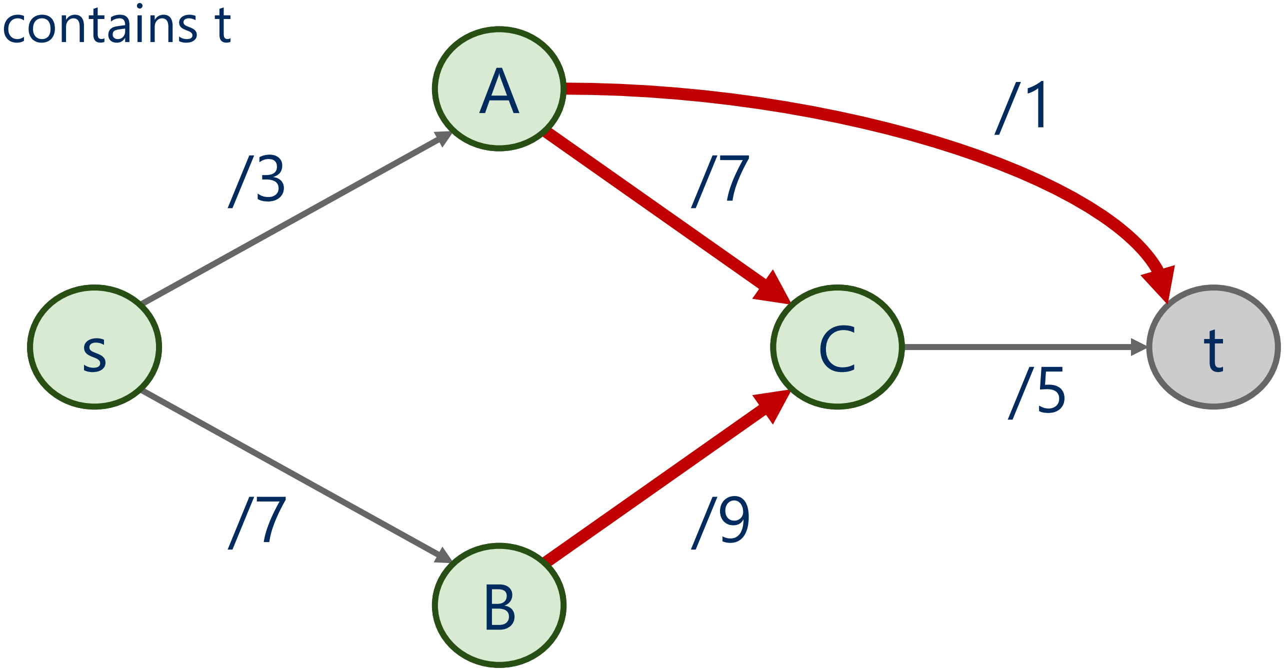
Finding bottleneck edges

- An **st-cut** on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t



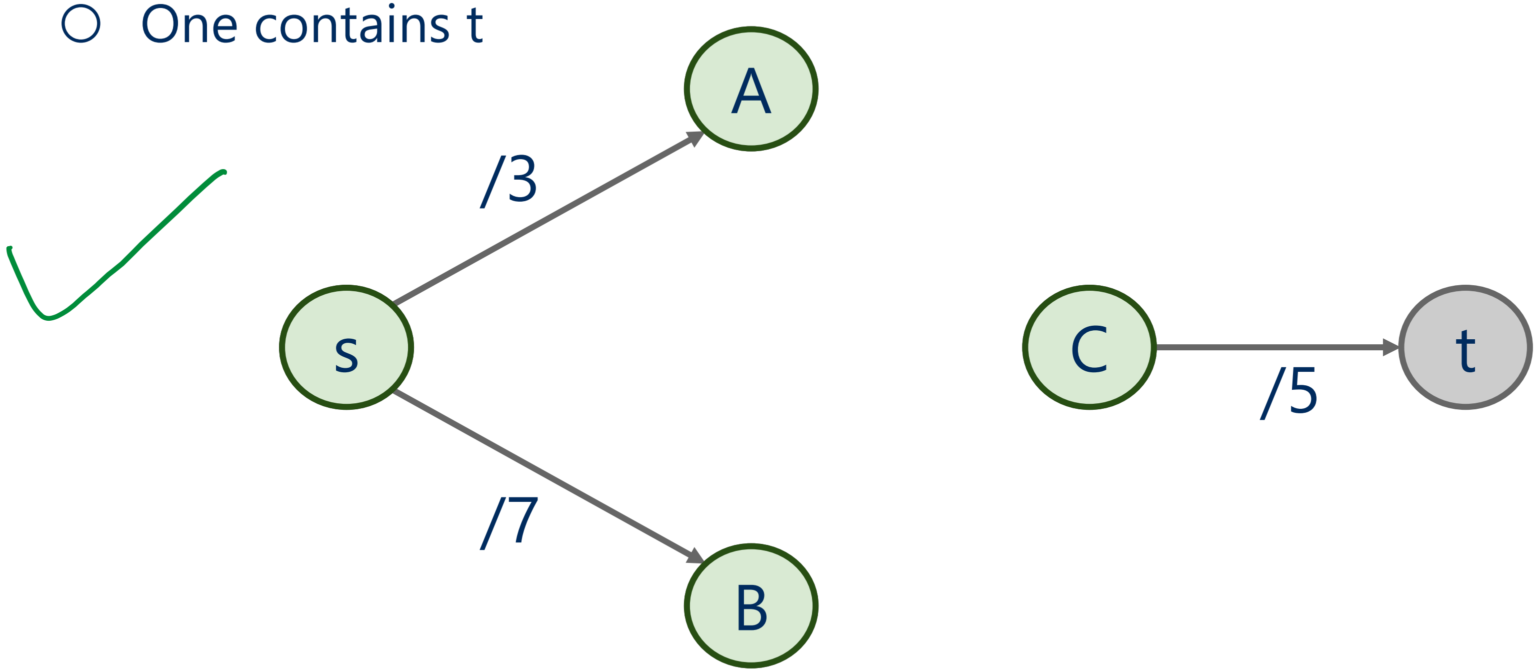
Finding bottleneck edges

- An **st-cut** on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t



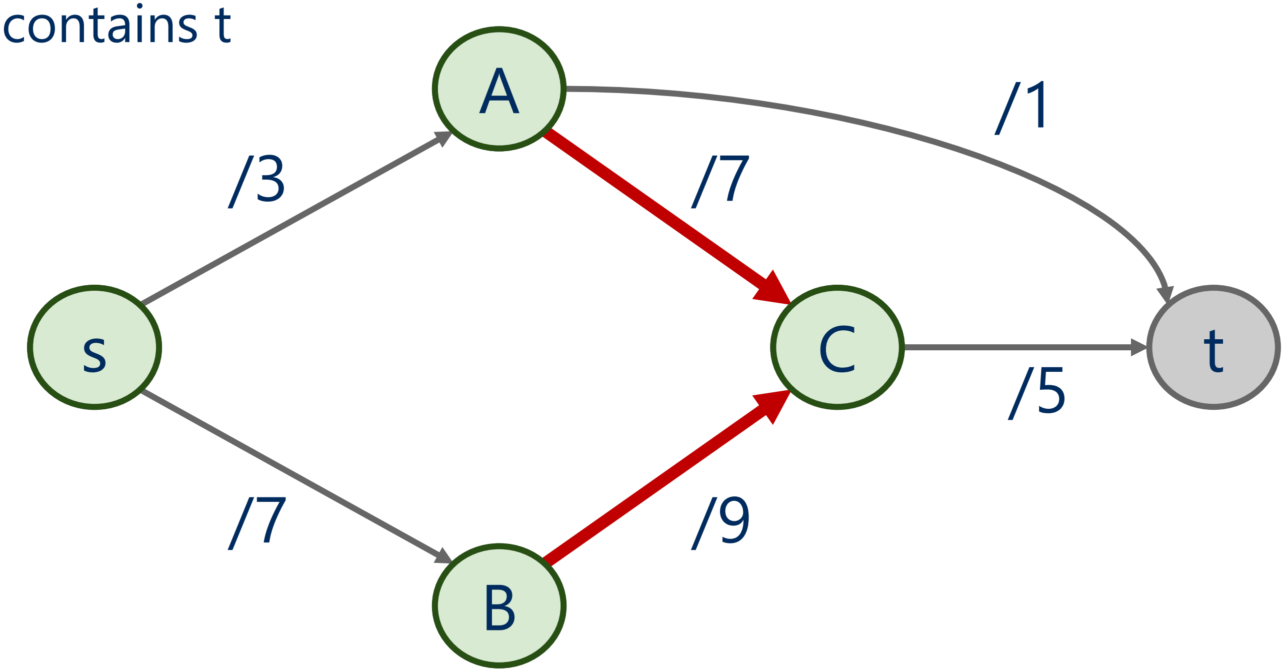
Finding bottleneck edges

- An **st-cut** on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t



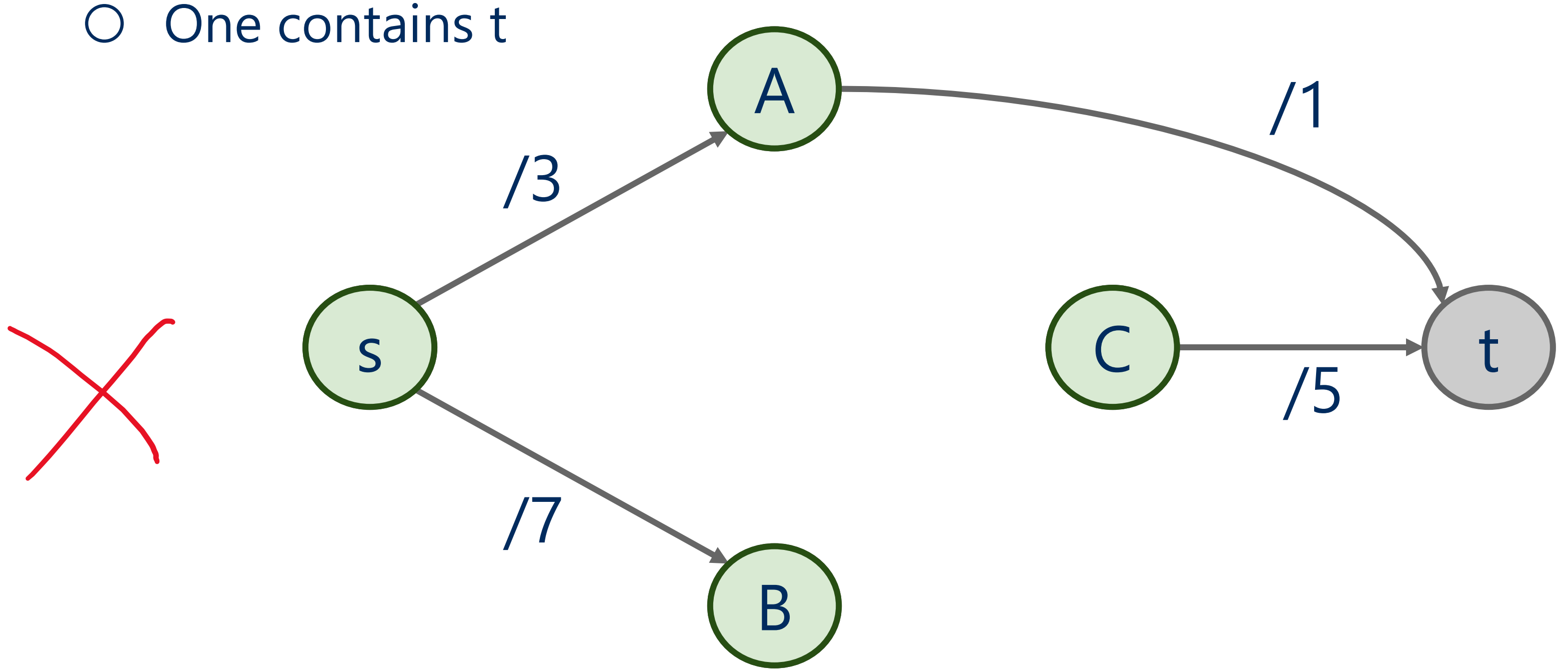
Finding bottleneck edges

- An **st-cut** on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t



Finding bottleneck edges

- An **st-cut** on G is a set of edges in G that, if removed, will partition the vertices of G into two disjoint sets
  - One contains s
  - One contains t



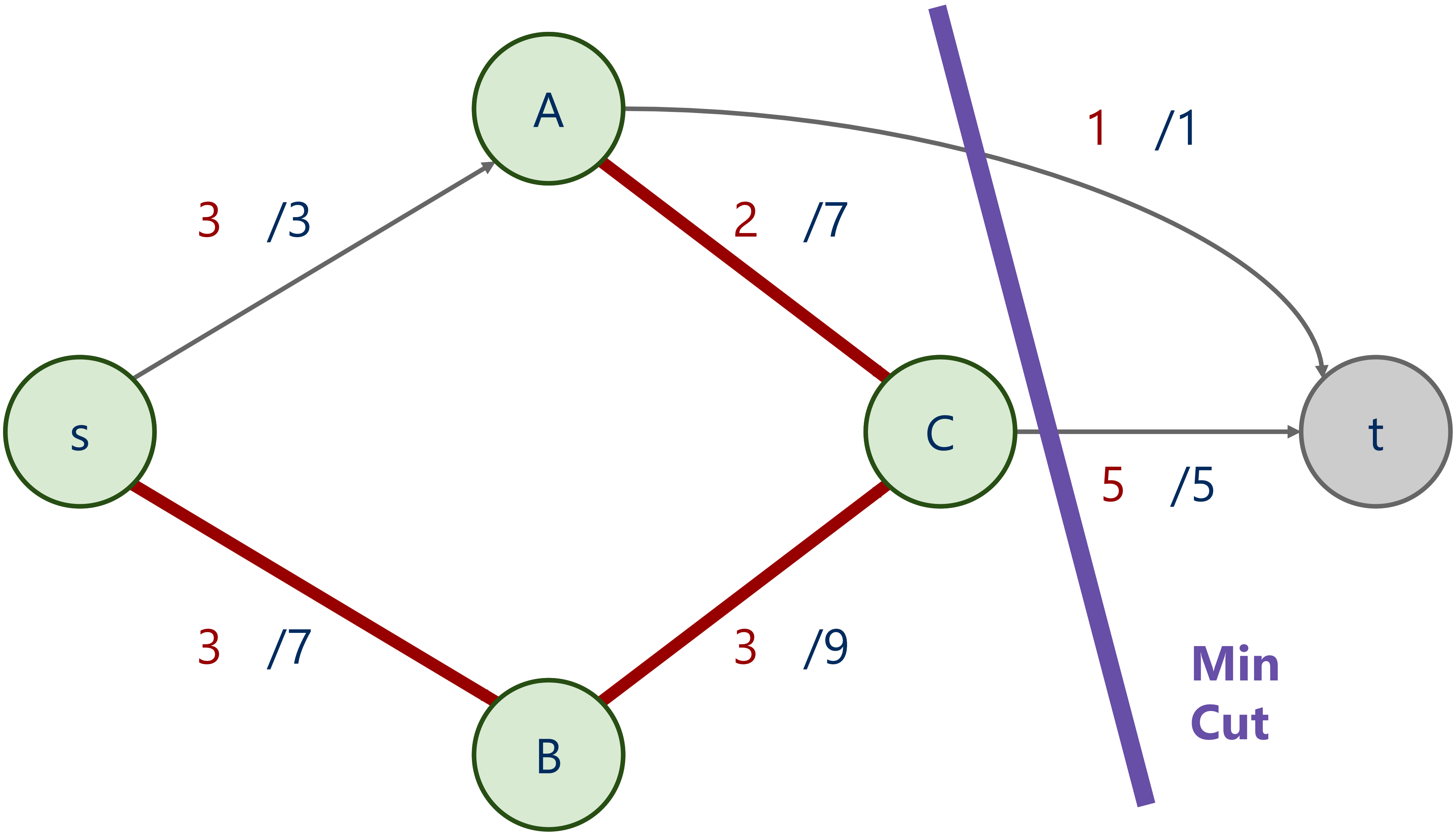
Finding bottleneck edges

- May be many st-cuts for a given graph
- Let’s focus on finding a **minimum st-cut**
  - an st-cut with the **smallest edge capacities**
  - May not be unique

How do we find a min st-cut?

- We could examine residual graphs
  - Specifically, try and allocate flow in the graph until we get to a residual graph with **no augmenting paths**
- The last iteration of Ford-Fulkerson visits every vertex reachable from s
  - Edges with only one endpoint reachable from S comprise a minimum st-cut

Determining the min cut



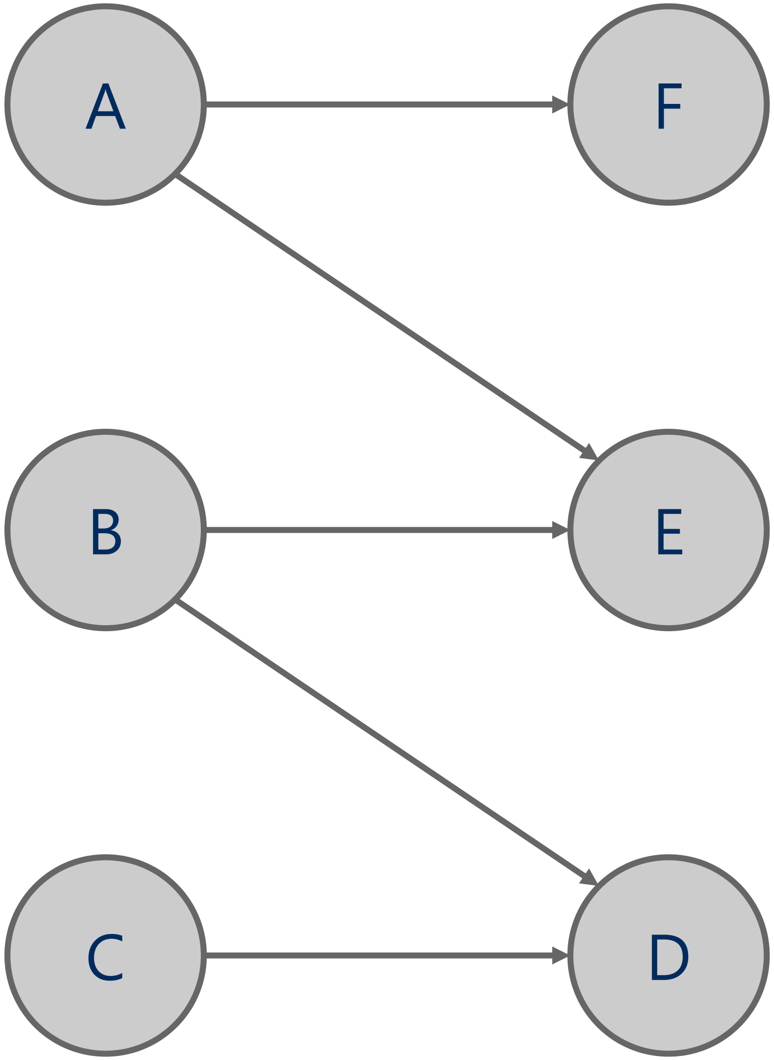
Max flow == Min cut

This is a special case of **duality**

- I.e., look at optimization problem from **two angles**
  - e.g., find the maximum flow or minimum cut
  - The difference between solution values referred to as **duality gap**
    - If the duality gap = 0, **strong duality** holds
      - Max flow/min cut uphold strong duality
    - If the duality gap > 0, **weak duality** holds

Maximum Bipartite Matching

Bipartite Graph: vertices decomposed into **two disjoint sets** such that no two vertices within the same set have an edge between them

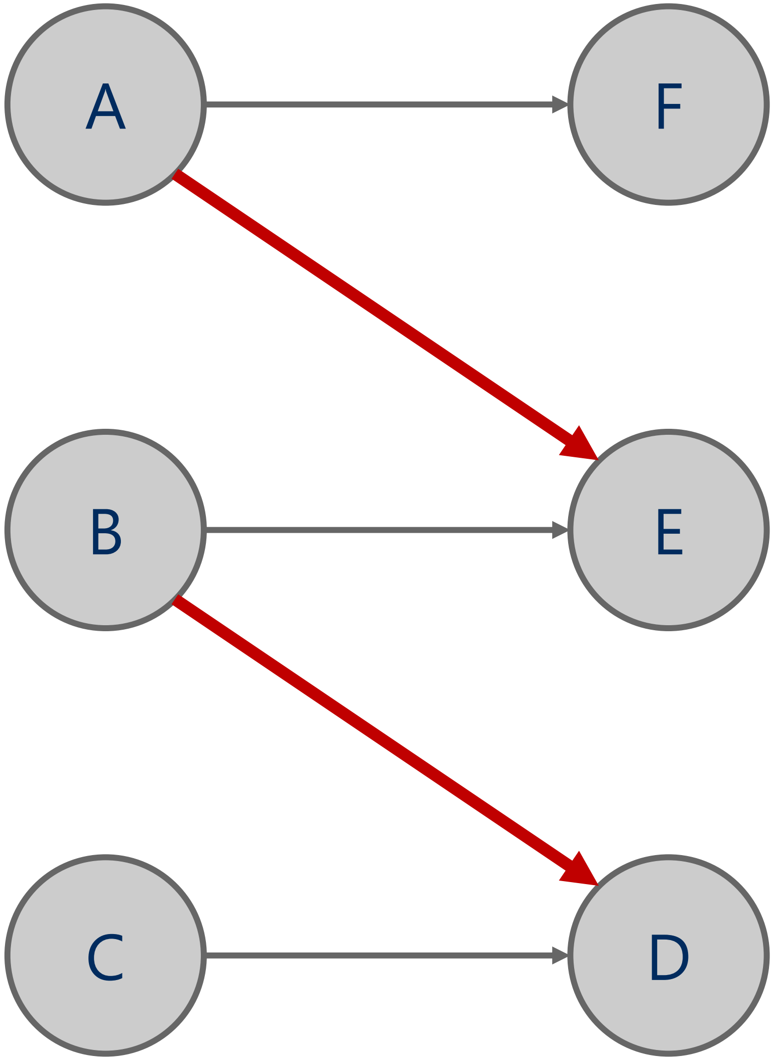


Example: computing tasks and machines

- certain tasks can only run on certain machines
- Run as many tasks as possible

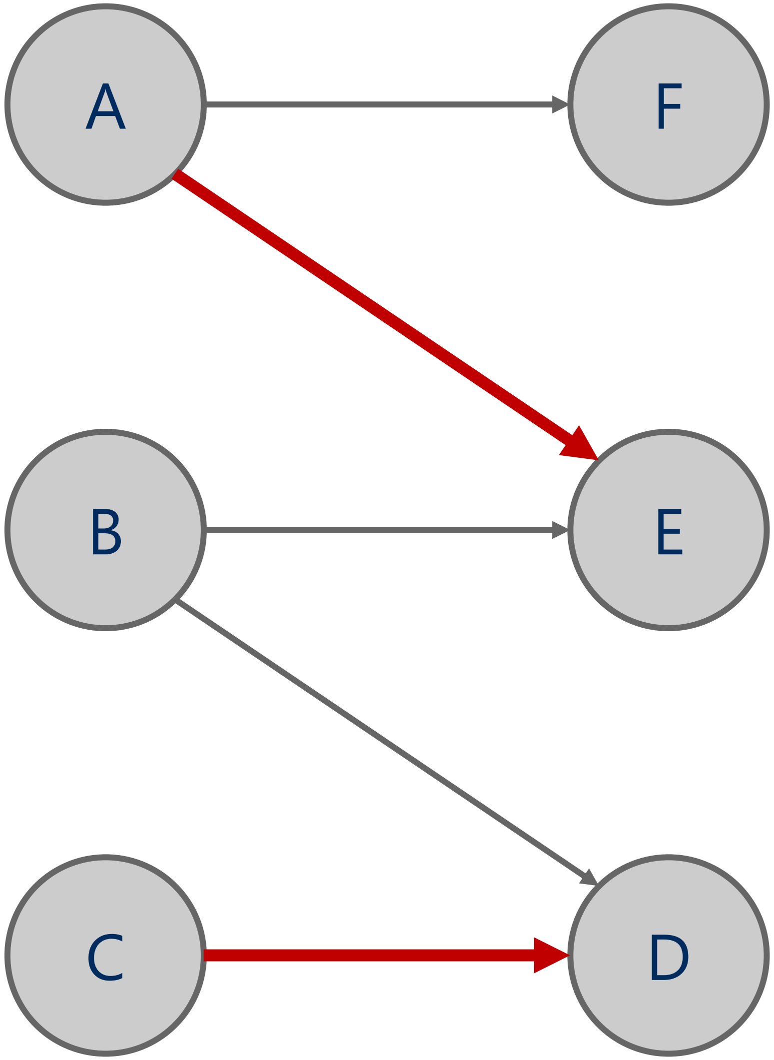
Maximum Bipartite Matching

Bipartite matching: a set of edges such that no two edges share an endpoint



Maximum Bipartite Matching

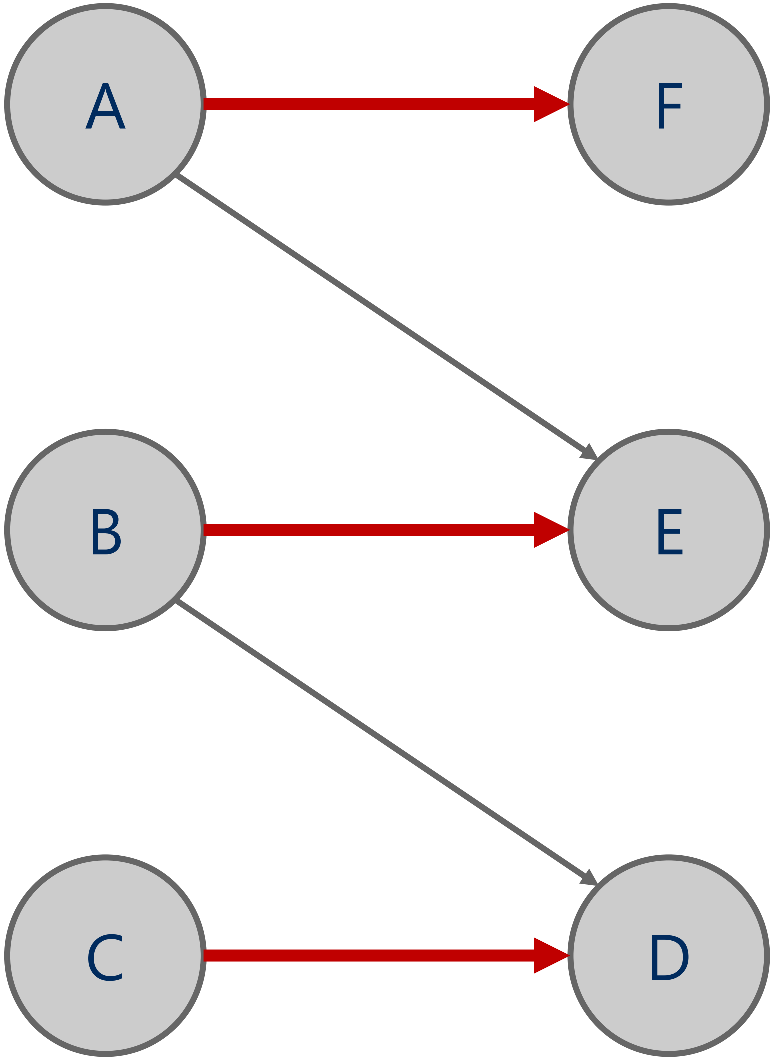
Bipartite matching: a set of edges such that no two edges share an endpoint



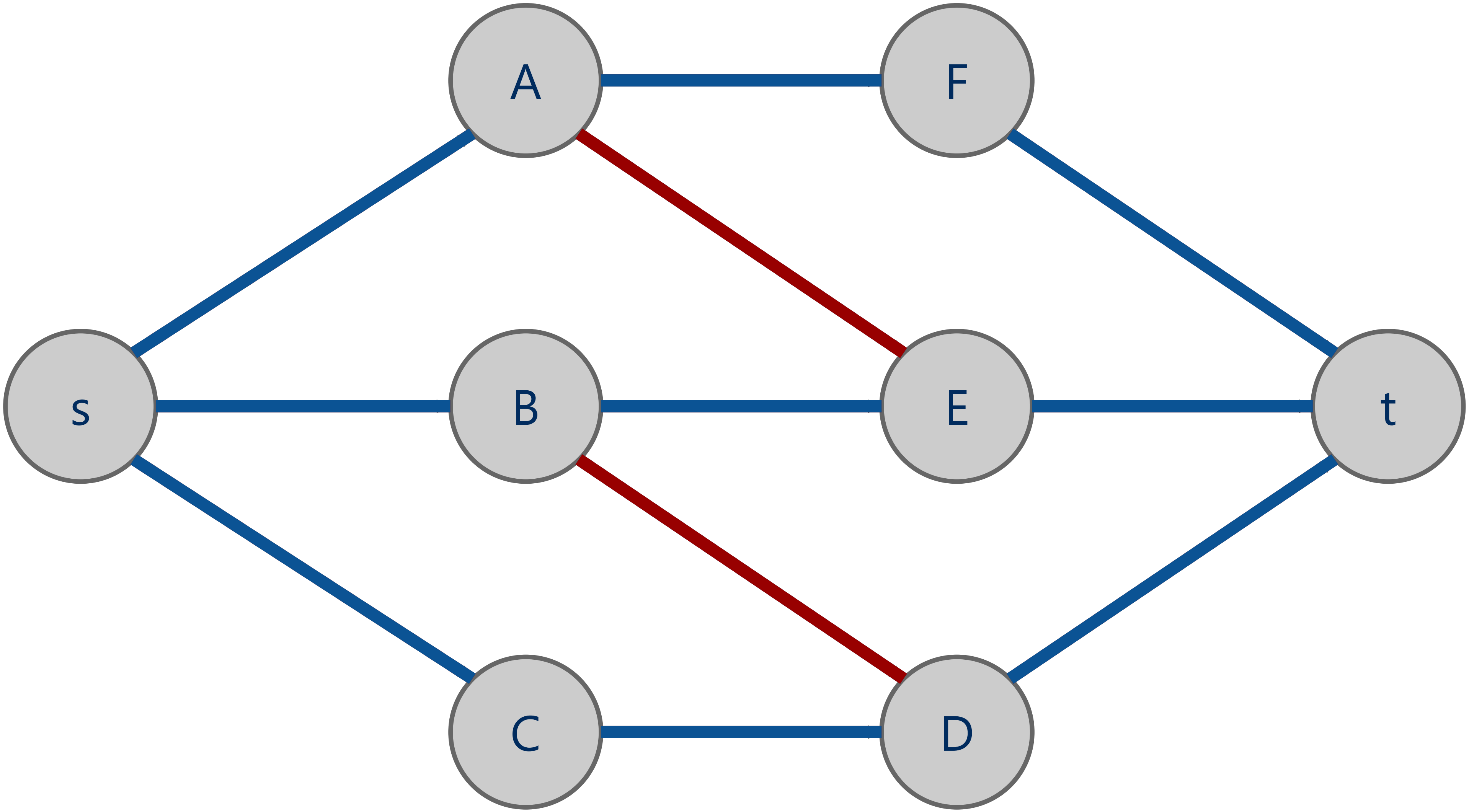
Maximum Bipartite Matching

Maximum matching: the largest possible matching

Let's **reduce** the problem to Maximum Flow!



Solving Maximum Bipartite Matching using Maximum Flow

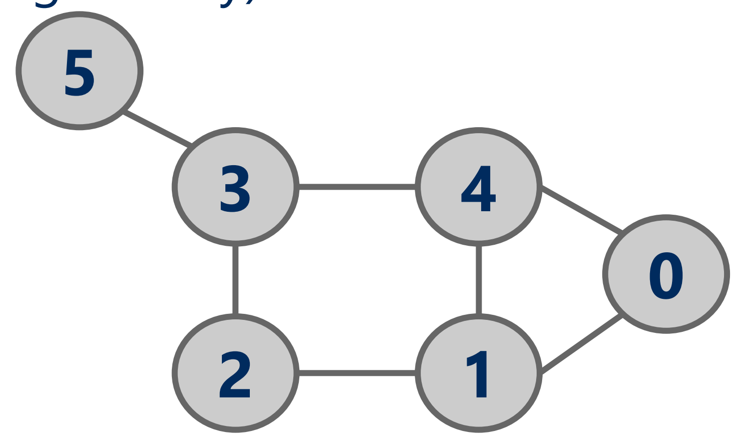


Graph Compression

- Real-life graphs are huge
  - 100’s if not 1000’s of GBs
  - Facebook graph, Google graph, maps, …
- Let’s see one (partial) idea for reducing the size of large graphs

Graph Compression

- **Step 1:** Construct a Compressed Sparse Row (CSR) representation of the graph
- CSR
  - Edges array concatenates ***sorted*** neighbor lists of all vertices
  - Offsets array:
    - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

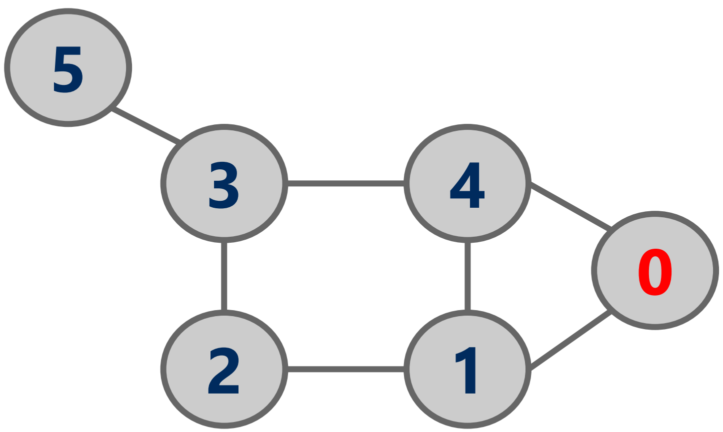
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

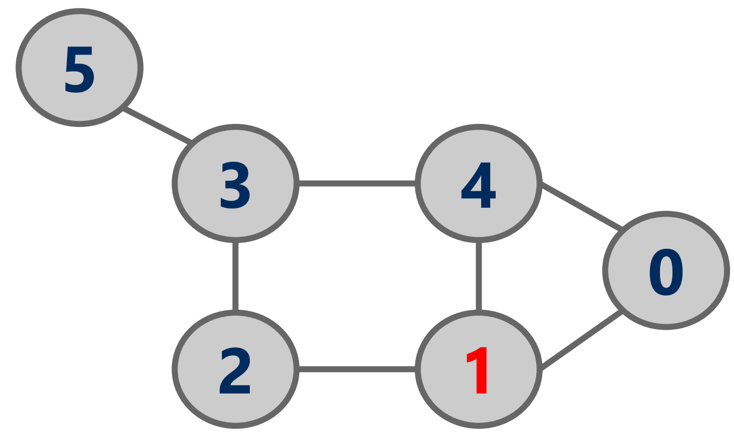
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
    - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

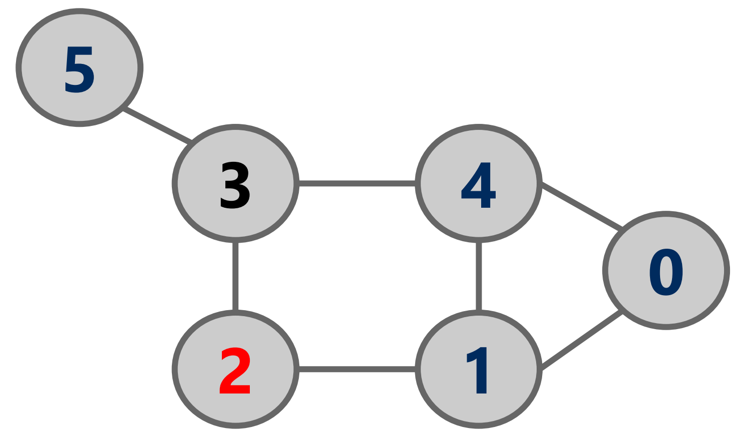
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let’s start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

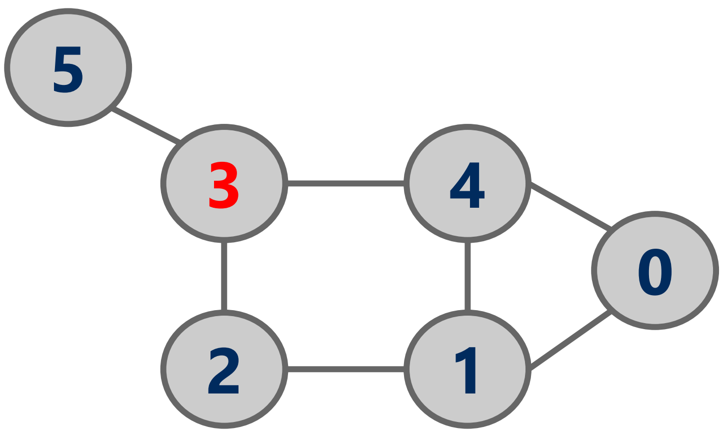
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

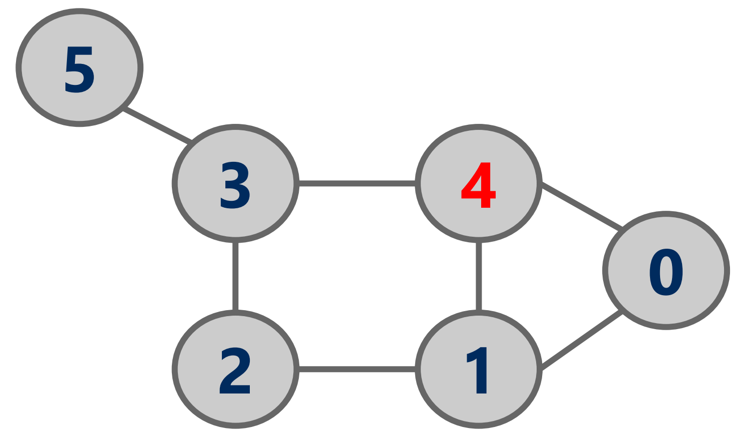
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Edges	1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let's start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

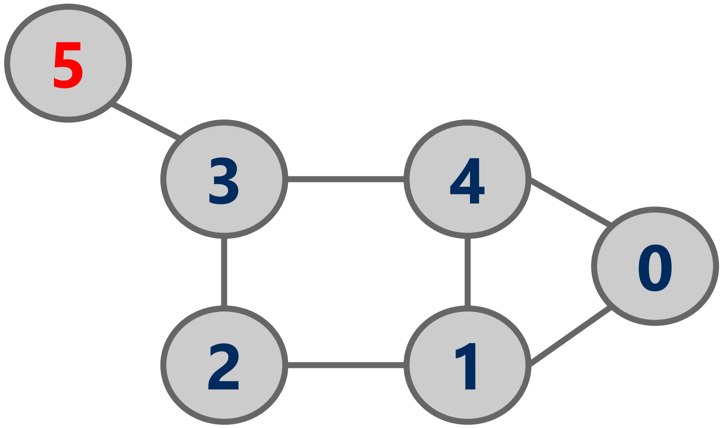
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Let’s start with one more graph representation
- Compressed Sparse Row (CSR)
- Edges array concatenates *sorted* neighbor lists of all vertices
- Offsets array:
  - offsets[v] is the starting index (in the Edges array) for the neighbors of vertex v



Edges

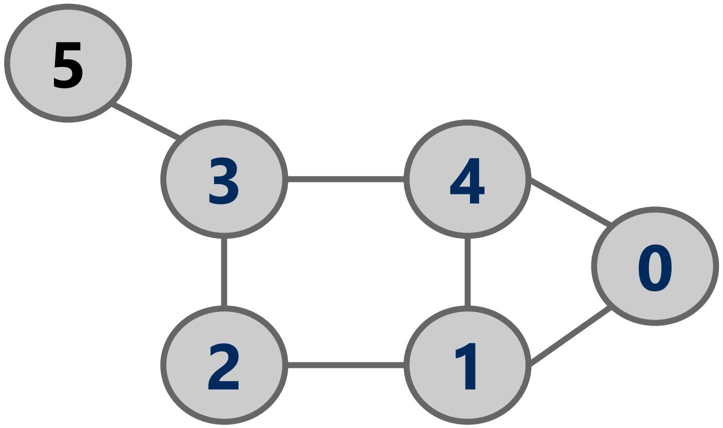
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- Can we compute the degree of a vertex using the offsets array?
  - Running time?
- What is the required space of this representation?
  - Theta(v + e)
  - Assume 4 bytes per vertex and per edge
  - Total size: *4*v* + *8*e* bytes



Edges

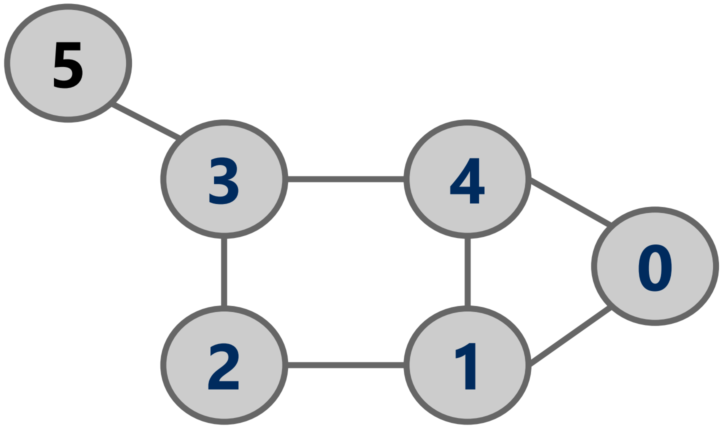
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list $v_1, v_2, v_3, \ldots$
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

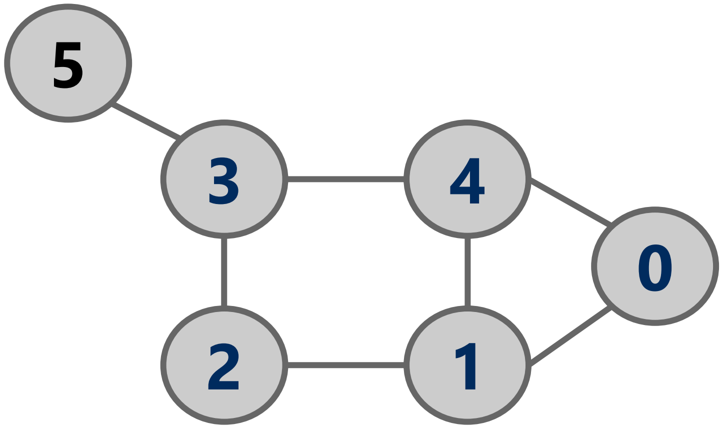
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v)$, $(v_2 - v_1)$, $(v_3 - v_2)$, …



Edges

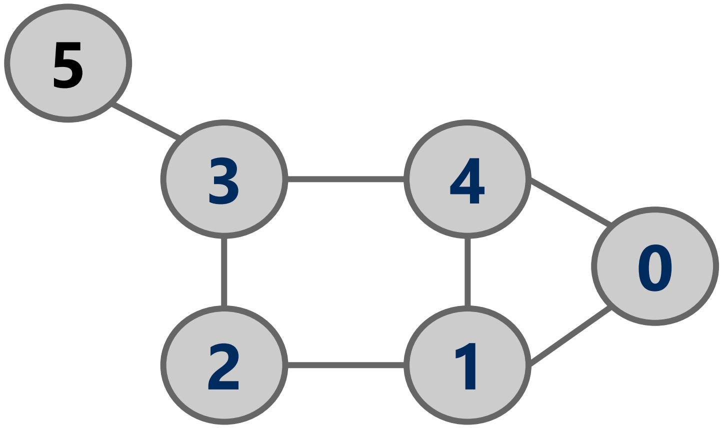
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1-0	4-1												

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, ...
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

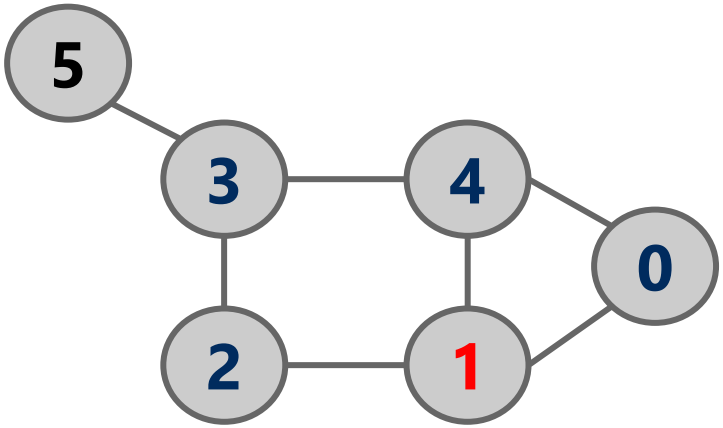
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3												

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

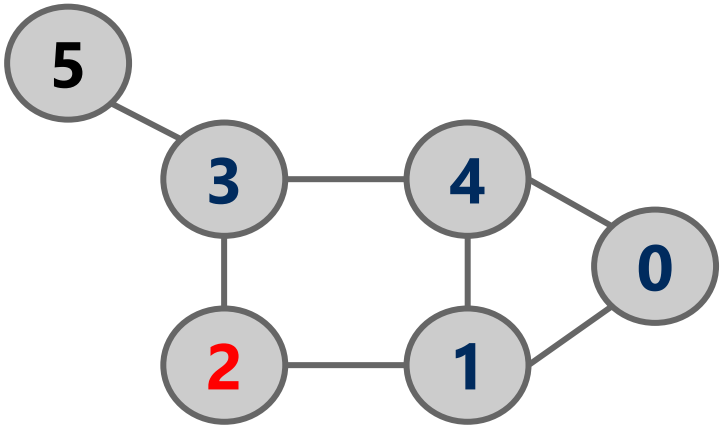
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2									

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

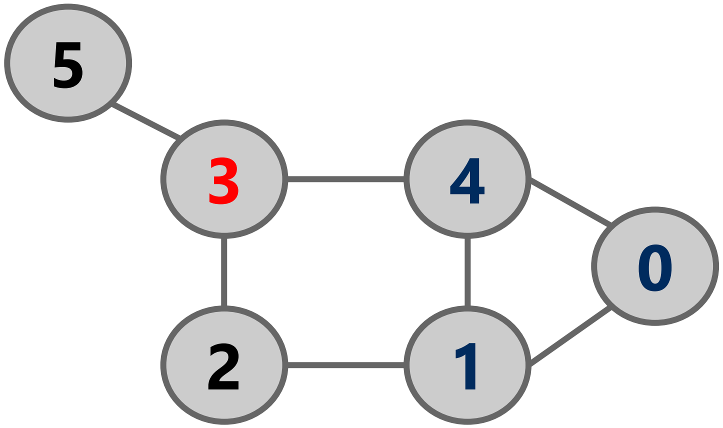
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2							

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, ...
  - Store the differences between each two consecutive numbers
    - $(v_1 - v)$, $(v_2 - v_1)$, $(v_3 - v_2)$, ...



Edges

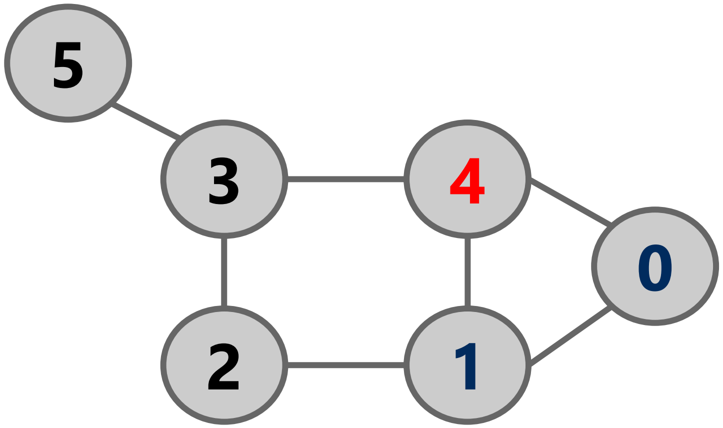
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1				

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, ...
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

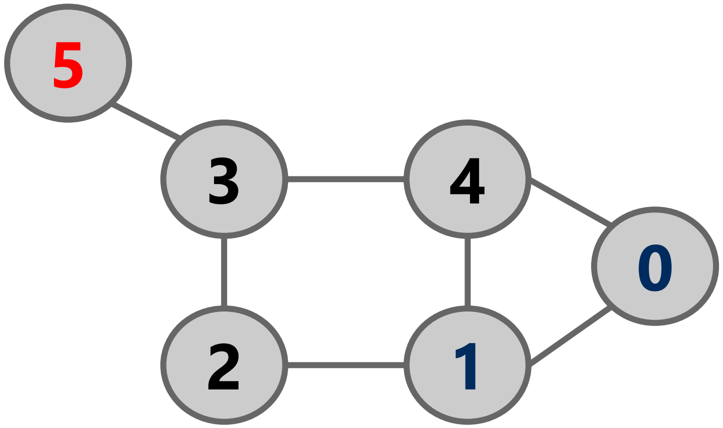
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

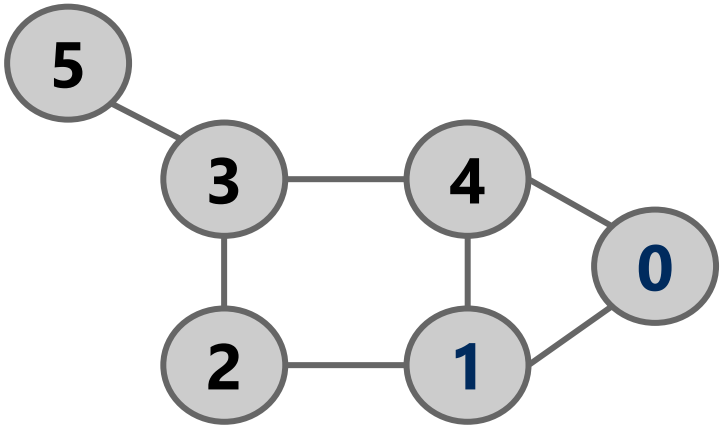
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	0	2	4	1	3	2	4	5	0	1	3	3
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Graph Compression

- **Step 2: Difference coding**
  - For each vertex *v*, with a neighbor list v_1, v_2, v_3, …
  - Store the differences between each two consecutive numbers
    - $(v_1 - v), (v_2 - v_1), (v_3 - v_2), \ldots$



Edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

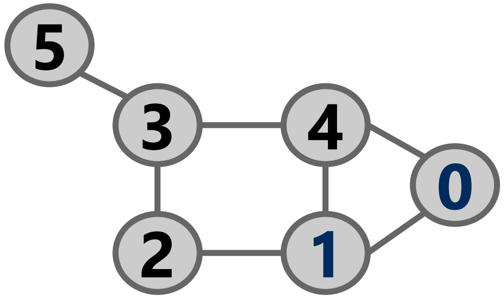
0	2	5	7	10	13

Graph Compression

- **Goal**: make the differences small
    - between vertex labels in each neighbor list small
- For Web Graphs
    - Each vertex is a web page
    - Sort the pages based on **reverse URL** (e.g., edu.pitt.cs.www)
    - Use sorted array indexes as vertex labels
    - Most links are local (within the same domain)
        - neighbors will be close to each other in the sorted list
        - Goal achieved!
- Other graphs can be relabeled to achieve that goal
    - https://www.cs.cmu.edu/~guyb/papers/BBK03.pdf

Graph Compression

- **Step 3: Use Gamma code to compress the differences**



Edges

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	3	-1	2	2	-1	2	-1	2	1	-4	1	2	-2

Offsets

0	2	5	7	10	13

Gamma Code

- Gamma Code: compress data when
  - small values much more **frequent** than large values
- To encode an integer *x*,
  - T = **largest power of 2 < *x***
  - Encode T as **(log T) zeros** followed by **1**
  - Append the remaining **(log T)** bits of **x – T**
- need **2*floor(log x) + 1** bits << 32 for small x

Gamma Code

- Example: Gamma encode 17: **10001**
  - $T = 16 = 2^4$
  - 1st part of Gamma code: **0000 1**
  - x – T = 17 – 16 = 0001
  - Gamma code: **00001 0001**

Clustering Problem

- **Input**: a set of **n data points** and a **target number of clusters, K**
- **Output**: K clusters
  - K cluster **centroids** (central points)
  - A **label** from the set {1, .., K} for each of the n data points
  - Sum of squared distances from each point to centroid is **minimum**

Clustering Problem

- minimize **distance** defined as:
- for(int i=0; i<n; i++)

  distance += (data[i] – centroid[cluster[i]]) ^{2}

Useful but hard problem!

- unsupervised machine learning algorithms
- dimensionality reduction problems
- **NP-hard**!
  - no efficient solution has been known yet
  - we don’t know yet if an efficient algorithm exists

Lloyd’s Local Search Algorithm

Initialization:

- Start with an **initial** cluster **assignment**
- Compute **initial cluster centroids**

Repeat until no change in centroids and cluster assignments

- Assign each data point to the closest cluster centroid
- Recompute cluster centroids based on new assignment

Limitation of Lloyd’s Algorithm

- Sensitive to initial clustering
- Fix: select initial centroids **as far from each other** as possible

K-Means ++ Algorithm for initial centroids

- Select first centroid with **uniform probability** over all data
- **Repeat** for each of the remaining K-1 initial centroids
  - For each data point
    - Compute distance to nearest centroid
  - Select next centroid with probability that favors data points with **larger distances**